PRINCIPLES OF COMPILER DESIGN

V RAGHAVAN

The McGraw·Hill Companies

Principles of Compiler Design

ABOUT THE AUTHOR

V Raghavan is a software developer by profession. Currently, he is working as a Technical Manager in Wipro Technologies, developing software in the technology domain. In the past, he has worked with Tata Consultancy Services and NCORE Technologies as a member of software development teams.

Raghavan has developed software for several leading international companies like AT&T, Analog Devices, Lucent Technologies, Hewlett-Packard, and Telstra Multimedia. He has designed and developed software in different domains—Embedded Systems, Wireless, Web based Solutions, Static Code Analysis, Networking and Test Automation, to name a few.

He is passionate about Compilers, Operating Systems and Artificial Intelligence. This book was written as a part of experiential learning while developing a Toy C Compiler in his spare time. He loves working on smart algorithms for solving tricky problems. He was one of the regular participants in Programmer of the Month (POTM) contest—an international programming contest at AT&T.

Raghavan holds an M.Tech in Computer Science and Engineering from Osmania University, Hyderabad, and a B.Tech in Mechanical Engineering from Jawaharlal Nehru Technological University, Hyderabad.

He can be reached through email at raghavan.compilers2009@yahoo.co.in.

The McGraw-Hill Companies

Principles of Compiler Design

V Raghavan
Technical Manager
Wipro Technologies, Bangalore

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

The McGraw-Hill Companies

Tata McGraw Hill

Published by the Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008.

Principles of Compiler Design

ISBN 13: 978-0-07-014471-2
ISBN 10: 0-07-014471-0

Managing Director: *Ajay Shukla*

Head—Higher Education Publishing: *Vibha Mahajan*
Manager: Sponsoring—SEM & Tech Ed: *Shalini Jha*
Editorial Executive: *Surabhi Shukla*
Development Editor: *Surbhi Suman*
Jr Executive—Editorial Services: *Dipika Dey*
Jr Manager—Production: *Anjali Razdan*

General Manager: Marketing—Higher Education: *Michael J Cruz*
Sr Product Manager—SEM & Tech Ed: *Biju Ganesan*
Asst Product Manager—SEM & Tech Ed: *Amit Paranjpe*

General Manager—Production: *Rajender P Ghansela*
Asst General Manager—Production: *B L Dogra*

Typeset at Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar-110 092 and printed at Pashupati Printers Pvt. Ltd., 1/429/16, Gali No. 1, Friends Colony, Industrial Area, G.T. Road, Shahdara, Delhi - 110 095

Cover Printer: Rashtriya Printers

RAXLCRBFDRRBQ

The McGraw-Hill Companies

Dedicated to

My Parents – G K Viswanathan and V Bhagirathy

My Wife – S Prathibha

My Little One – R Pranav

CONTENTS

PREFACE

Compilers are utilities that transform programs written in higher level languages like Pascal, C, and C++ into lower level languages like the assembly language program or the machine code. The machine code can be directly executed on a computer to perform various tasks. The assembly language program can be converted to machine code by using another utility called assembler and then executed on a computer.

This book describes the internals of a compiler detailing the steps used by it to transform a higher level language program into a lower language program.

Purpose

In my 18 years of industrial experience, I have had the opportunity to absorb a number of fresh computer science graduates in my team. My observation has been that the fresh computer science graduates get a theoretical level of understanding of the compilers and its functions during their course work. These young men and women often find it difficult to get into job assignments that deal with compilers and the associated tools. They usually have to be trained to make them ready for the job assignments involving compilers. One of the reasons for this gap is lack of textbooks that emphasise on practical application of principles of compilers like Syntax Analysis, Semantic Analysis, etc. This book bridges the gap and provides students of computer science good reading material to understand the basics of compilers.

During my survey of textbooks on compilers, I found that there was a dearth of textbooks that had the right mix of theory and practice for compiler construction. I did not find many textbooks in the market that focused on introducing compilers to the entry-level student. Most of the available textbooks were suitable for an advanced course in compiler construction rather than the introductory course. Most of the existing textbooks also had a heavy theoretical emphasis. I embarked on writing this book because I felt a genuine need for a textbook that had (a) focus on the basics of compilers (b) accent on the practical aspect and (c) suitability to an entry-level student.

Background

It was the summer of 2001 that I was contemplating on writing a book in the field of computer science as my service to the industry that gave me bread and butter. I was more inclined towards writing a book on the subject of Artificial Intelligence because (a) I had implemented many of the AI algorithms while competing at the Programmer of the Month (POTM) contest, during my stay in AT&T Paradyne at NJ, USA. (b) I was reasonably convinced that students would find my book encouraging, since they would see "Source Code" and not just Theory.

On 18 May 2001, I spoke to Prof. Khodanpur, Retired Head of the department of computer science at RV College of Engineering, Bangalore, on the subject of writing a book on Artificial Intelligence. He encouraged me to undertake writing a book on Principles of Compiler Design instead of Artificial Intelligence. Prof. Khodanpur impressed upon me that the students were struggling to get a grasp on the basics of compilers and it would do a world of good, if I can contribute something to help the students. That was the genesis of this book on the principles of compilers. Prof. Khodanpur subsequently reviewed my first output and gave me some suggestions for improvement. I have tried my best to keep up to his philosophy of 'keeping it simple and correct' throughout this book.

For 8 years, I have been relentlessly working on various aspects of this book. The Almighty chose me and bestowed upon me the strength and the means to carry on this incredible journey. The blessings of my parents helped me overcome many a challenge during the process of making this book. The inspiration, happiness and purpose came from my constant companion and buddy—Pranav. There were numerous times in this span of 8 years, where I felt that I cannot go any further and thought of giving it all up. Fortunately, God willed it otherwise and today I live to see my dream fulfilled. It has been a fantastic experience writing this book and I hope that you find reading the book equally pleasurable.

Target Readers

The textbook is intended for an introductory level student who is familiar with C/C++ Programming. The goal is to cover the basics of the compiler theory rather than being an exhaustive complete reference in the subject. The textbook would be ideal for graduate/undergraduate level of students taking an introductory course on the compilers. It is particularly well suited for undergraduate students of computer science and information technology engineering. This book can also be used by professionals who wish to understand the basics of compilers in order to prepare themselves for working on projects based on compilers.

Salient Features

The salient features of the book are

- Simple chapter organisation based on different stages of a compiler
- Easy narrative style of explanation with emphasis on basic principles
- Numerous examples and illustrations clarifying concepts
- Incremental development of a Toy C language compiler
- References to the behaviour of production compilers

Organisation

The textbook is organised into 7 chapters. The first chapter provides an introduction to compilers. It talks about the different utilities that participate in the compilation process. It details the functionality of a compiler, describing briefly each stage of the compilation. This forms the basis for the rest of the book.
The different stages of compilation are described in successive chapters starting from Chapter 2. Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation and Target Code generation are the topics for discussions in Chapters 2,3,4,5 and 6 respectively.

The techniques for optimising the intermediate code and the target code are described in Chapter 7—Optimisation.

The approach taken in each of the chapters is to introduce the theory with suitable practical examples. Algorithm description is usually followed by an implementation and demonstrated by an example. A number of examples use the C language compiler of GNU's compiler collection (gcc) as a reference to illustrate the behaviour of the compilers.

Online Learning Centre

The book needs to be read along with source code for examples that can be downloaded from http://www.mhhe.com/raghavan/pcd. These examples have been compiled and tested on CYGWIN 1.5 platform on my home x86 PC running Win98. I have also checked the examples on LINUX platform. I have used gcc 3.4, flex 2.5 and bison 2.3 for compiling and testing the examples. An older version of gcc, namely gcc-2.95 was used in one of the examples of Chapter 1 to illustrate the compilation process.

The reader should install CYGWIN 1.5 (or newer) for compiling and checking out the source code of examples in the case of PC running Windows. The source code can be compiled and checked on Personal Computers running LINUX Operating System. An HTML based documentation for the source code is also available as a part of the download.

Supplements for Instructors: PowerPoint slides, class-test quizzes with answers, chapter-wise references, and lab assignments.

Supplements for Students: Chapter-wise tutorials and self-test quizzes with answers.

The Yahoo Groups at http://in.groups.yahoo.com/group/compilers2009 is a useful place to post any issues with regard to the compilation/execution of the source code, specific to your Operating System.

Acknowledgements

I am grateful to Prof Khodanpur, Retired Head of the computer science department at RV college of Engineering, Bangalore, for giving me the idea to write a book on compilers and reviewing my first output. I hope I have done justice to his expectations.

I am indebted to my family for all the sacrifices they had to make in order to fulfill my dream. Thank you Amma and Little fellow for everything. You have been a wonderful family. This book would not have been possible without your support, love and affection.

I would like to acknowledge the help of several people who have made a difference in my life. I owe a Big Thank you to Dr Aswath N Rao, N R Ramesh, M K Suresh Kumar, Priya Suresh, K Srinivas, R Sundararajan, J Swaminathan, Siby Abraham, S Krishna Kumar, Tinku Jose, Vijay Garapati, Sanjay Khodanpur, Shashikala, Srinath Rajaram, A Ramasamy, M Vijay Anand, V Balachandran, David Lobaccaro, Prakash Viswanathan and Srinivasan.

I would like to thank my first employers—Vinay Deshpande and Shashank Garg at NCORE Technologies, Bangalore—for giving me an entry into the software field. I want to thank my colleagues and friends at Wipro Technologies for all their help and cooperation.

I also want to thank Vibha Mahajan, Shalini Jha, Nilanjan Chakravarty, Surabhi Shukla, Surbhi Suman, Dipika Dey, Anjali Razdan and Baldev Raj at Tata McGraw Hill Education for their help and guidance in the course of making this book. A note of acknowledgement is due to the following reviewers for their valuable feedback.

Preeti Aggarwal
Punjab University of Technology (earlier Punjab Engineering College), Chandigarh

M Tripathi
Institute of Engineering & Technology, Kanpur

Shailendra Singh
Punjab University of Technology (earlier Punjab Engineering College), Chandigarh

Shalini Batra
Thapar Institute of Engineering & Technology, Patiala

K Umamaheshwari
PSG College of Technology, Coimbatore

R Prabhakar
Coimbatore Institute of Technology, Coimbatore

B N S Murthy
Nagarjuna College of Engineering & Technology, Bangalore

B I Khodanpur
R.V. College of Engineering, Bangalore

B K Sarkar
Birla Institute of Technology (BIT), Ranchi

The readers of the book are encouraged to send their comments, queries and suggestions at the following email id—tmh.csefeedback@gmail.com (*kindly mention the title and author name in the subject line*).

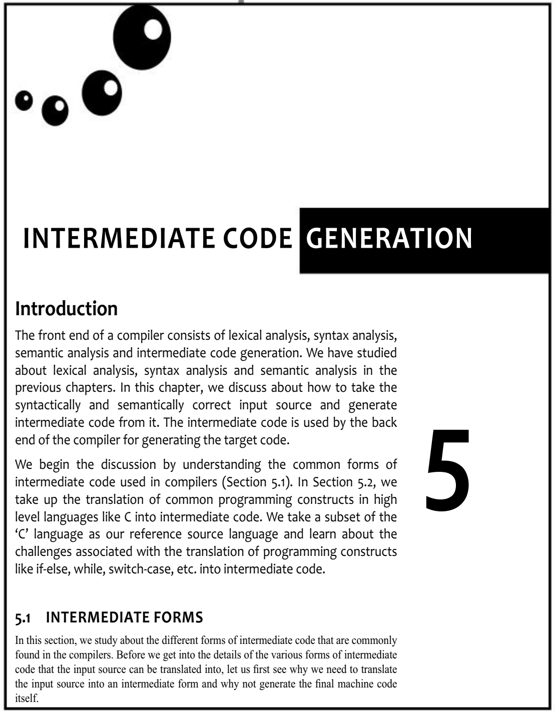
V Raghavan
raghavan.compilers2009@yahoo.co.in

The McGraw-Hill Companies

VISUAL WALKTHROUGH

INTRODUCTION

Each chapter begins with an Introduction that gives a summary of the background and the organisation of chapter's contents.



INTERMEDIATE CODE GENERATION

5

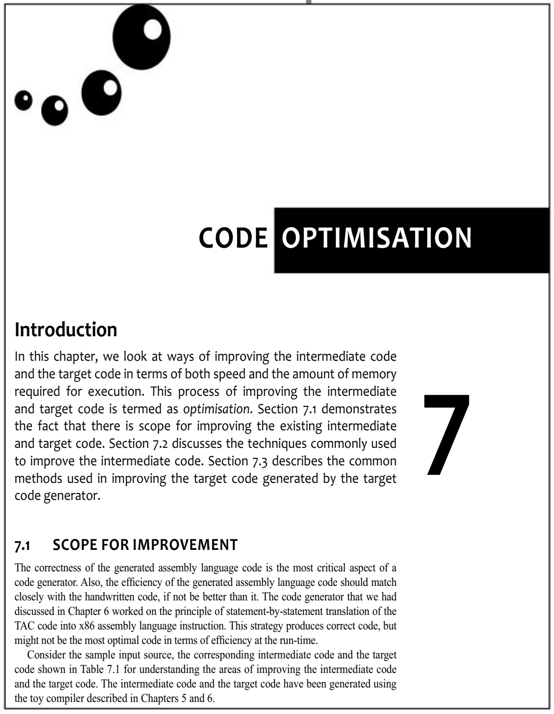
Introduction

The front end of a compiler consists of lexical analysis, syntax analysis, semantic analysis and intermediate code generation. We have studied about lexical analysis, syntax analysis and semantic analysis in the previous chapters. In this chapter, we discuss about how to take the syntactically and semantically correct input source and generate intermediate code from it. The intermediate code is used by the back end of the compiler for generating the target code.

We begin the discussion by understanding the common forms of intermediate code used in compilers (Section 5.1). In Section 5.2, we take up the translation of common programming constructs in high level languages like C into intermediate code. We take a subset of the 'C' language as our reference source language and learn about the challenges associated with the translation of programming constructs like if-else, while, switch-case, etc. into intermediate code.

5.1 INTERMEDIATE FORMS

In this section, we study about the different forms of intermediate code that are commonly found in the compilers. Before we get into the details of the various forms of intermediate code that the input source can be translated into, let us first see why we need to translate the input source into an intermediate form and why not generate the final machine code itself.



CODE OPTIMISATION

7

Introduction

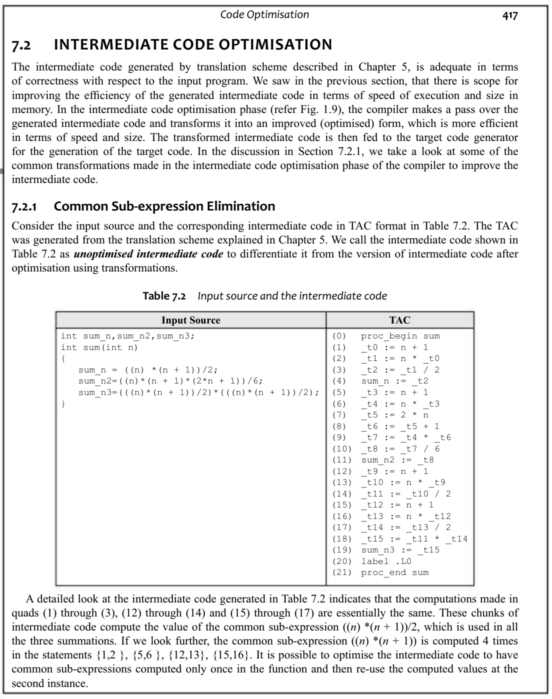
In this chapter, we look at ways of improving the intermediate code and the target code in terms of both speed and the amount of memory required for execution. This process of improving the intermediate and target code is termed as *optimisation*. Section 7.1 demonstrates the fact that there is scope for improving the existing intermediate and target code. Section 7.2 discusses the techniques commonly used to improve the intermediate code. Section 7.3 describes the common methods used in improving the target code generated by the target code generator.

7.1 SCOPE FOR IMPROVEMENT

The correctness of the generated assembly language code is the most critical aspect of a code generator. Also, the efficiency of the generated assembly language code should match closely with the handwritten code, if not be better than it. The code generator that we had discussed in Chapter 6 worked on the principle of statement-by-statement translation of the TAC code into x86 assembly language instruction. This strategy produces correct code, but might not be the most optimal code in terms of efficiency at the run-time.

Consider the sample input source, the corresponding intermediate code and the target code shown in Table 7.1 for understanding the areas of improving the intermediate code and the target code. The intermediate code and the target code have been generated using the toy compiler described in Chapters 5 and 6.

The McGraw-Hill Companies



7.2 INTERMEDIATE CODE OPTIMISATION

The intermediate code generated by translation scheme described in Chapter 5, is adequate in terms of correctness with respect to the input program. We saw in the previous section, that there is scope for improving the efficiency of the generated intermediate code in terms of speed of execution and size in memory. In the intermediate code optimisation phase (refer Fig. 1.9), the compiler makes a pass over the generated intermediate code and transforms it into an improved (optimised) form, which is more efficient in terms of speed and size. The transformed intermediate code is then fed to the target code generator for the generation of the target code. In the discussion in Section 7.2.1, we take a look at some of the common transformations made in the intermediate code optimisation phase of the compiler to improve the intermediate code.

7.2.1 Common Sub-expression Elimination

Consider the input source and the corresponding intermediate code in TAC format in Table 7.2. The TAC was generated from the translation scheme explained in Chapter 5. We call the intermediate code shown in Table 7.2 as ***unoptimised intermediate code*** to differentiate it from the version of intermediate code after optimisation using transformations.

Table 7.2 *Input source and the intermediate code*

Input Source	TAC

`int sum_n,sum_n2,sum_n3;`
`int sum(int n)`
`{`
`   sum_n = ((n) *(n + 1))/2;`
`   sum_n2=((n)*(n + 1)*(2*n + 1))/6;`
`   sum_n3=(((n)*(n + 1))/2)*(((n)*(n + 1))/2);`
`}`
	`(0)  proc_begin sum`
	`(1)  _t0 := n + 1`
	`(2)  _t1 := n * _t0`
	`(3)  _t2 := _t1 / 2`
	`(4)  sum_n := _t2`
	`(5)  _t3 := n + 1`
	`(6)  _t4 := n * _t3`
	`(7)  _t5 := 2 * n`
	`(8)  _t6 := _t5 + 1`
	`(9)  _t7 := _t4 * _t6`
	`(10) _t8 := _t7 / 6`
	`(11) sum_n2 := _t8`
	`(12) _t9 := n + 1`
	`(13) _t10 := n * _t9`
	`(14) _t11 := _t10 / 2`
	`(15) _t12 := n + 1`
	`(16) _t13 := n * _t12`
	`(17) _t14 := _t13 / 2`
	`(18) _t15 := _t11 * _t14`
	`(19) sum_n3 := _t15`
	`(20) label .L0`
	`(21) proc_end sum`

A detailed look at the intermediate code generated in Table 7.2 indicates that the computations made in quads (1) through (3), (12) through (14) and (15) through (17) are essentially the same. These chunks of intermediate code compute the value of the common sub-expression $((n) *(n + 1))/2$, which is used in all the three summations. If we look further, the common sub-expression $((n) *(n + 1))$ is computed 4 times in the statements {1,2 }, {5,6 }, {12,13}, {15,16}. It is possible to optimise the intermediate code to have common sub-expressions computed only once in the function and then re-use the computed values at the second instance.

SECTIONS AND SUB-SECTIONS

Neatly divided into sections and sub-sections, the subject matter can be studied in a logical progression of ideas and concepts.

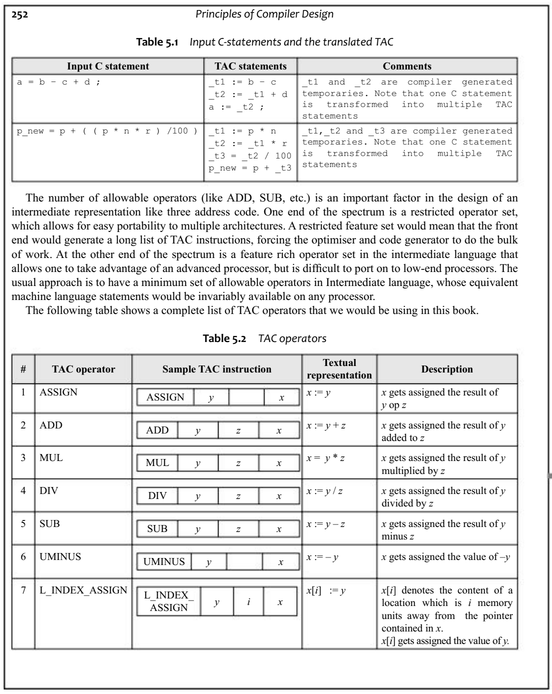


Table 5.1 *Input C-statements and the translated TAC*

Input C statement	TAC statements	Comments

`a = b - c + d ;`	`_t1 := b - c`
	`_t2 := _t1 + d`
	`a := _t2 ;`
		`_t1 and _t2 are compiler generated temporaries. Note that one C statement is transformed into multiple TAC statements`

`p_new = p + ( ( p * n * r ) /100 )`	`_t1 := p * n`
	`_t2 := _t1 * r`
	`_t3 = _t2 / 100`
	`p_new = p + _t3`
		`_t1,_t2 and _t3 are compiler generated temporaries. Note that one C statement is transformed into multiple TAC statements`

The number of allowable operators (like ADD, SUB, etc.) is an important factor in the design of an intermediate representation like three address code. One end of the spectrum is a restricted operator set, which allows for easy portability to multiple architectures. A restricted feature set would mean that the front end would generate a long list of TAC instructions, forcing the optimiser and code generator to do the bulk of work. At the other end of the spectrum is a feature rich operator set in the intermediate language that allows one to take advantage of an advanced processor, but is difficult to port on to low-end processors. The usual approach is to have a minimum set of allowable operators in Intermediate language, whose equivalent machine language statements would be invariably available on any processor.

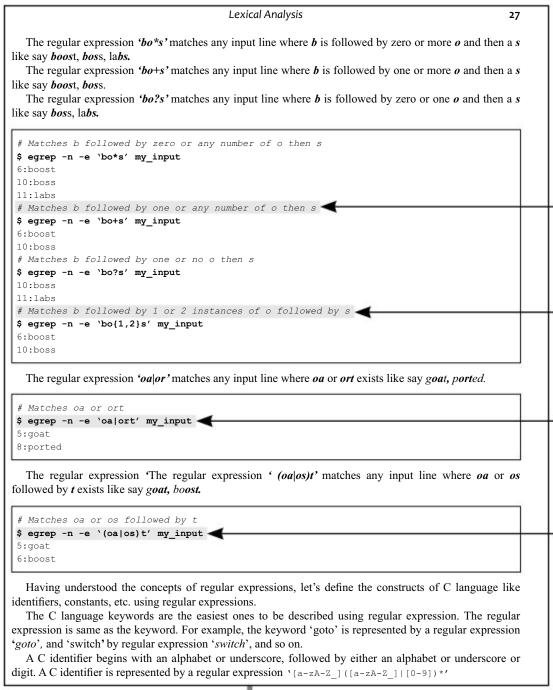
The following table shows a complete list of TAC operators that we would be using in this book.

Table 5.2 *TAC operators*

#	TAC operator	Sample TAC instruction	Textual representation	Description
1	ASSIGN	ASSIGN y x	$x := y$	x gets assigned the result of y op z
2	ADD	ADD y z x	$x := y + z$	x gets assigned the result of y added to z
3	MUL	MUL y z x	$x = y * z$	x gets assigned the result of y multiplied by z
4	DIV	DIV y z x	$x := y / z$	x gets assigned the result of y divided by z
5	SUB	SUB y z x	$x := y - z$	x gets assigned the result of y minus z
6	UMINUS	UMINUS y x	$x := -y$	x gets assigned the value of $-y$
7	L_INDEX_ASSIGN	L_INDEX_ ASSIGN y i x	$x[i] := y$	$x[i]$ denotes the content of a location which is i memory units away from the pointer contained in x. $x[i]$ gets assigned the value of y.

TABLES

Tables are provided in each chapter to aid in understanding of the text material.



Lexical Analysis **27**

The regular expression ***'bo*s'*** matches any input line where ***b*** is followed by zero or more ***o*** and then a ***s*** like say ***boos**t*, ***bos**s*, la***bs.***

The regular expression ***'bo+s'*** matches any input line where ***b*** is followed by one or more ***o*** and then a ***s*** like say ***boos**t*, ***bos**s*.

The regular expression ***'bo?s'*** matches any input line where ***b*** is followed by zero or one ***o*** and then a ***s*** like say ***bos**s*, la***bs.***

```
# Matches b followed by zero or any number of o then s
$ egrep -n -e 'bo*s' my_input
6:boost
10:boss
11:labs
# Matches b followed by one or any number of o then s
$ egrep -n -e 'bo+s' my_input
6:boost
10:boss
# Matches b followed by one or no o then s
$ egrep -n -e 'bo?s' my_input
10:boss
11:labs
# Matches b followed by 1 or 2 instances of o followed by s
$ egrep -n -e 'bo{1,2}s' my_input
6:boost
10:boss
```

The regular expression ***'oa|or'*** matches any input line where ***oa*** or ***ort*** exists like say *goat, ported.*

```
# Matches oa or ort
$ egrep -n -e 'oa|ort' my_input
5:goat
8:ported
```

The regular expression 'The regular expression ' ***(oa|os)t'*** matches any input line where ***oa*** or ***os*** followed by ***t*** exists like say *goat, boost.*

```
# Matches oa or os followed by t
$ egrep -n -e '(oa|os)t' my_input
5:goat
6:boost
```

Having understood the concepts of regular expressions, let's define the constructs of C language like identifiers, constants, etc. using regular expressions.

The C language keywords are the easiest ones to be described using regular expression. The regular expression is same as the keyword. For example, the keyword 'goto' is represented by a regular expression '*goto*', and 'switch' by regular expression '*switch*', and so on.

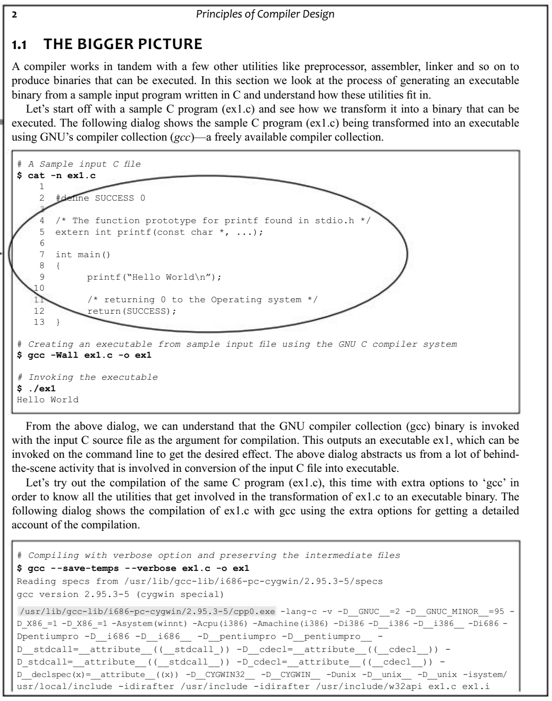
A C identifier begins with an alphabet or underscore, followed by either an alphabet or underscore or digit. A C identifier is represented by a regular expression `'[a-zA-Z_]([a-zA-Z_]|[0-9])*'`

Comments for Reader's Understanding in *Italics*

User Commands in **Bold**

DIALOGS

Dialogs showing the human-computer interactions are provided in each chapter to help the readers appreciate the relevant practical aspects.



2 *Principles of Compiler Design*

1.1 THE BIGGER PICTURE

A compiler works in tandem with a few other utilities like preprocessor, assembler, linker and so on to produce binaries that can be executed. In this section we look at the process of generating an executable binary from a sample input program written in C and understand how these utilities fit in.

Let's start off with a sample C program (ex1.c) and see how we transform it into a binary that can be executed. The following dialog shows the sample C program (ex1.c) being transformed into an executable using GNU's compiler collection (*gcc*)—a freely available compiler collection.

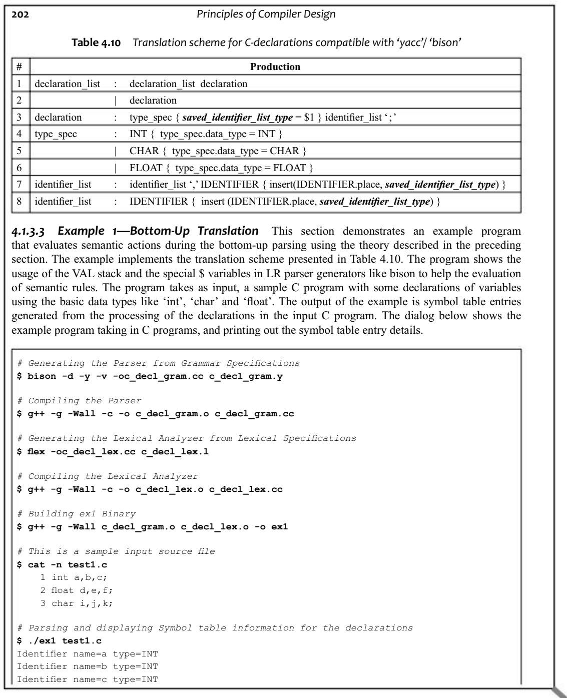
```
# A Sample input C file
$ cat -n ex1.c
     1
     2  #define SUCCESS 0
     3
     4  /* The function prototype for printf found in stdio.h */
     5  extern int printf(const char *, ...);
     6
     7  int main()
     8  {
     9       printf("Hello World\n");
    10
    11       /* returning 0 to the Operating system */
    12       return(SUCCESS);
    13  }

# Creating an executable from sample input file using the GNU C compiler system
$ gcc -Wall ex1.c -o ex1

# Invoking the executable
$ ./ex1
Hello World
```

From the above dialog, we can understand that the GNU compiler collection (gcc) binary is invoked with the input C source file as the argument for compilation. This outputs an executable ex1, which can be invoked on the command line to get the desired effect. The above dialog abstracts us from a lot of behind-the-scene activity that is involved in conversion of the input C file into executable.

Let's try out the compilation of the same C program (ex1.c), this time with extra options to 'gcc' in order to know all the utilities that get involved in the transformation of ex1.c to an executable binary. The following dialog shows the compilation of ex1.c with gcc using the extra options for getting a detailed account of the compilation.

```
# Compiling with verbose option and preserving the intermediate files
$ gcc --save-temps --verbose ex1.c -o ex1
Reading specs from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/specs
gcc version 2.95.3-5 (cygwin special)
 /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/cpp0.exe -lang-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -
D_X86_=1 -D_X86_=1 -Asystem(winnt) -Acpu(i386) -Amachine(i386) -Di386 -D__i386 -D__i386__ -Di686 -
Dpentiumpro -D__i686 -D__i686__ -D__pentiumpro -D__pentiumpro__ -
D__stdcall=__attribute__((__stdcall__)) -D__cdecl=__attribute__((__cdecl__)) -
D_stdcall=__attribute__((__stdcall__)) -D_cdecl=__attribute__((__cdecl__)) -
D__declspec(x)=__attribute__((x)) -D__CYGWIN32__ -D__CYGWIN__ -Dunix -D__unix__ -D__unix -isystem/
usr/local/include -idirafter /usr/include -idirafter /usr/include/w32api ex1.c ex1.i
```

The output from the computer in regular font.



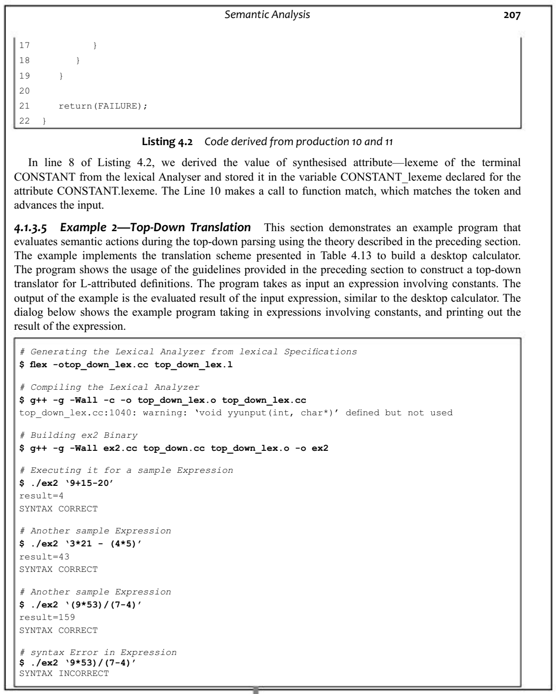
202 *Principles of Compiler Design*

Table 4.10 *Translation scheme for C-declarations compatible with 'yacc'/ 'bison'*

#	Production		
1	declaration_list	:	declaration_list declaration
2		|	declaration
3	declaration	:	type_spec { ***saved_identifier_list_type*** = $1 } identifier_list ';'
4	type_spec	:	INT { type_spec.data_type = INT }
5		|	CHAR { type_spec.data_type = CHAR }
6		|	FLOAT { type_spec.data_type = FLOAT }
7	identifier_list	:	identifier_list ',' IDENTIFIER { insert(IDENTIFIER.place, ***saved_identifier_list_type***) }
8	identifier_list	:	IDENTIFIER { insert (IDENTIFIER.place, ***saved_identifier_list_type***) }

4.1.3.3 Example 1—Bottom-Up Translation This section demonstrates an example program that evaluates semantic actions during the bottom-up parsing using the theory described in the preceding section. The example implements the translation scheme presented in Table 4.10. The program shows the usage of the VAL stack and the special $ variables in LR parser generators like bison to help the evaluation of semantic rules. The program takes as input, a sample C program with some declarations of variables using the basic data types like 'int', 'char' and 'float'. The output of the example is symbol table entries generated from the processing of the declarations in the input C program. The dialog below shows the example program taking in C programs, and printing out the symbol table entry details.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -oc_decl_gram.cc c_decl_gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o c_decl_gram.o c_decl_gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc_decl_lex.cc c_decl_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o c_decl_lex.o c_decl_lex.cc

# Building ex1 Binary
$ g++ -g -Wall c_decl_gram.o c_decl_lex.o -o ex1

# This is a sample input source file
$ cat -n test1.c
    1 int a,b,c;
    2 float d,e,f;
    3 char i,j,k;

# Parsing and displaying Symbol table information for the declarations
$ ./ex1 test1.c
Identifier name=a type=INT
Identifier name=b type=INT
Identifier name=c type=INT
```



Semantic Analysis 207

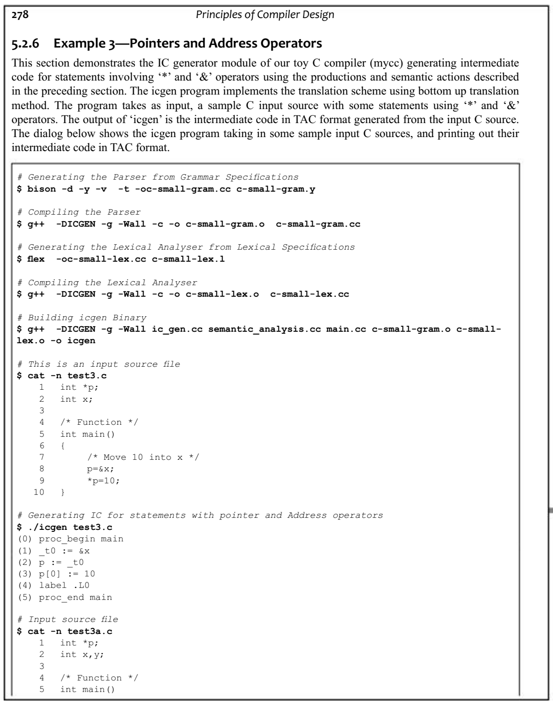
```
17            }
18         }
19     }
20
21     return(FAILURE);
22  }
```

Listing 4.2 *Code derived from production 10 and 11*

In line 8 of Listing 4.2, we derived the value of synthesised attribute—lexeme of the terminal CONSTANT from the lexical Analyser and stored it in the variable CONSTANT_lexeme declared for the attribute CONSTANT.lexeme. The Line 10 makes a call to function match, which matches the token and advances the input.

4.1.3.5 Example 2—Top-Down Translation This section demonstrates an example program that evaluates semantic actions during the top-down parsing using the theory described in the preceding section. The example implements the translation scheme presented in Table 4.13 to build a desktop calculator. The program shows the usage of the guidelines provided in the preceding section to construct a top-down translator for L-attributed definitions. The program takes as input an expression involving constants. The output of the example is the evaluated result of the input expression, similar to the desktop calculator. The dialog below shows the example program taking in expressions involving constants, and printing out the result of the expression.

```
# Generating the Lexical Analyzer from lexical Specifications
$ flex -otop_down_lex.cc top_down_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o top_down_lex.o top_down_lex.cc
top_down_lex.cc:1040: warning: 'void yyunput(int, char*)' defined but not used

# Building ex2 Binary
$ g++ -g -Wall ex2.cc top_down.cc top_down_lex.o -o ex2

# Executing it for a sample Expression
$ ./ex2 '9+15-20'
result=4
SYNTAX CORRECT

# Another sample Expression
$ ./ex2 '3*21 - (4*5)'
result=43
SYNTAX CORRECT

# Another sample Expression
$ ./ex2 '(9*53)/(7-4)'
result=159
SYNTAX CORRECT

# syntax Error in Expression
$ ./ex2 '9*53)/(7-4)'
SYNTAX INCORRECT
```



278 *Principles of Compiler Design*

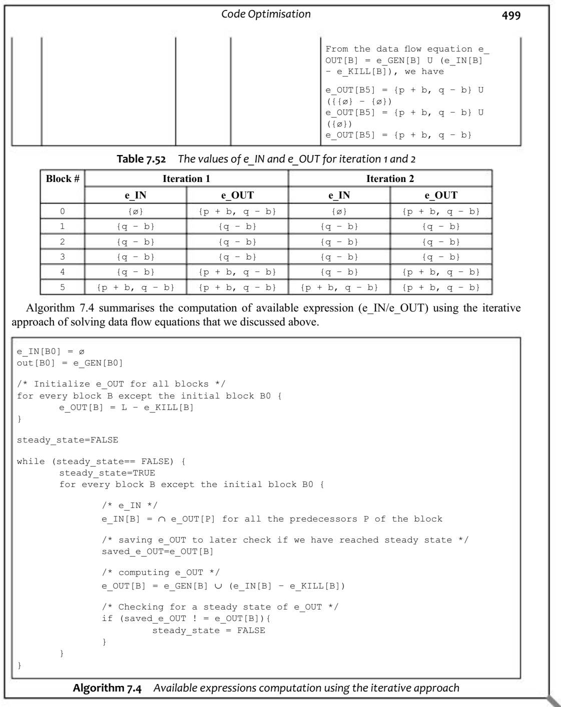
5.2.6 Example 3—Pointers and Address Operators

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving '*' and '&' operators using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom up translation method. The program takes as input, a sample C input source with some statements using '*' and '&' operators. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# This is an input source file
$ cat -n test3.c
    1   int *p;
    2   int x;
    3
    4   /* Function */
    5   int main()
    6   {
    7        /* Move 10 into x */
    8        p=&x;
    9        *p=10;
   10   }

# Generating IC for statements with pointer and Address operators
$ ./icgen test3.c
(0) proc_begin main
(1) _t0 := &x
(2) p := _t0
(3) p[0] := 10
(4) label .L0
(5) proc_end main

# Input source file
$ cat -n test3a.c
    1   int *p;
    2   int x,y;
    3
    4   /* Function */
    5   int main()
```

SAMPLE CODE IMPLEMENTATION

A Dialog pertaining to a sample implementation immediately follows the theory to reinforce the ideas correctly. All the source code used in the textbook is online and can be downloaded from the website http://www.mhhe.com/raghavan/pcd.



				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]), we have
				e_OUT[B5] = {p + b, q – b} U ({{ø} – {ø})
				e_OUT[B5] = {p + b, q – b} U ({ø})
				e_OUT[B5] = {p + b, q – b}

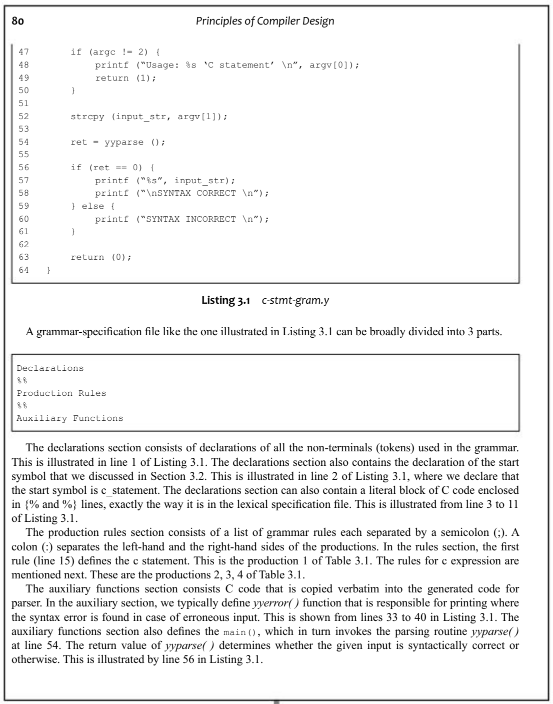
Table 7.52 *The values of e_IN and e_OUT for iteration 1 and 2*

Block #	Iteration 1		Iteration 2	
	e_IN	e_OUT	e_IN	e_OUT
0	{ø}	{p + b, q – b}	{ø}	{p + b, q – b}
1	{q – b}	{q – b}	{q – b}	{q – b}
2	{q – b}	{q – b}	{q – b}	{q – b}
3	{q – b}	{q – b}	{q – b}	{q – b}
4	{q – b}	{p + b, q – b}	{q – b}	{p + b, q – b}
5	{p + b, q – b}	{p + b, q – b}	{p + b, q – b}	{p + b, q – b}

Algorithm 7.4 summarises the computation of available expression (e_IN/e_OUT) using the iterative approach of solving data flow equations that we discussed above.

```
e_IN[B0] = ø
out[B0] = e_GEN[B0]

/* Initialize e_OUT for all blocks */
for every block B except the initial block B0 {
        e_OUT[B] = L – e_KILL[B]
}

steady_state=FALSE

while (steady_state== FALSE) {
        steady_state=TRUE
        for every block B except the initial block B0 {

                /* e_IN */
                e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block

                /* saving e_OUT to later check if we have reached steady state */
                saved_e_OUT=e_OUT[B]

                /* computing e_OUT */
                e_OUT[B] = e_GEN[B] ∪ (e_IN[B] – e_KILL[B])

                /* Checking for a steady state of e_OUT */
                if (saved_e_OUT ! = e_OUT[B]){
                         steady_state = FALSE
                }
        }
}
```

Algorithm 7.4 *Available expressions computation using the iterative approach*



```
47        if (argc != 2) {
48            printf ("Usage: %s 'C statement' \n", argv[0]);
49            return (1);
50        }
51
52        strcpy (input_str, argv[1]);
53
54        ret = yyparse ();
55
56        if (ret == 0) {
57            printf ("%s", input_str);
58            printf ("\nSYNTAX CORRECT \n");
59        } else {
60            printf ("SYNTAX INCORRECT \n");
61        }
62
63        return (0);
64    }
```

Listing 3.1 *c-stmt-gram.y*

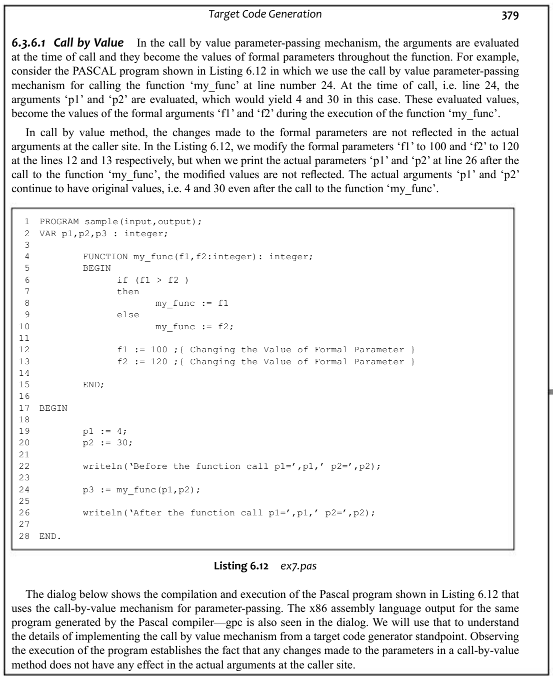
A grammar-specification file like the one illustrated in Listing 3.1 can be broadly divided into 3 parts.

```
Declarations
%%
Production Rules
%%
Auxiliary Functions
```

The declarations section consists of declarations of all the non-terminals (tokens) used in the grammar. This is illustrated in line 1 of Listing 3.1. The declarations section also contains the declaration of the start symbol that we discussed in Section 3.2. This is illustrated in line 2 of Listing 3.1, where we declare that the start symbol is c_statement. The declarations section can also contain a literal block of C code enclosed in {% and %} lines, exactly the way it is in the lexical specification file. This is illustrated from line 3 to 11 of Listing 3.1.

The production rules section consists of a list of grammar rules each separated by a semicolon (;). A colon (:) separates the left-hand and the right-hand sides of the productions. In the rules section, the first rule (line 15) defines the c statement. This is the production 1 of Table 3.1. The rules for c expression are mentioned next. These are the productions 2, 3, 4 of Table 3.1.

The auxiliary functions section consists C code that is copied verbatim into the generated code for parser. In the auxiliary section, we typically define *yyerror( )* function that is responsible for printing where the syntax error is found in case of erroneous input. This is shown from lines 33 to 40 in Listing 3.1. The auxiliary functions section also defines the `main()`, which in turn invokes the parsing routine *yyparse( )* at line 54. The return value of *yyparse( )* determines whether the given input is syntactically correct or otherwise. This is illustrated by line 56 in Listing 3.1.



6.3.6.1 Call by Value In the call by value parameter-passing mechanism, the arguments are evaluated at the time of call and they become the values of formal parameters throughout the function. For example, consider the PASCAL program shown in Listing 6.12 in which we use the call by value parameter-passing mechanism for calling the function 'my_func' at line number 24. At the time of call, i.e. line 24, the arguments 'p1' and 'p2' are evaluated, which would yield 4 and 30 in this case. These evaluated values, become the values of the formal arguments 'f1' and 'f2' during the execution of the function 'my_func'.

In call by value method, the changes made to the formal parameters are not reflected in the actual arguments at the caller site. In the Listing 6.12, we modify the formal parameters 'f1' to 100 and 'f2' to 120 at the lines 12 and 13 respectively, but when we print the actual parameters 'p1' and 'p2' at line 26 after the call to the function 'my_func', the modified values are not reflected. The actual arguments 'p1' and 'p2' continue to have original values, i.e. 4 and 30 even after the call to the function 'my_func'.

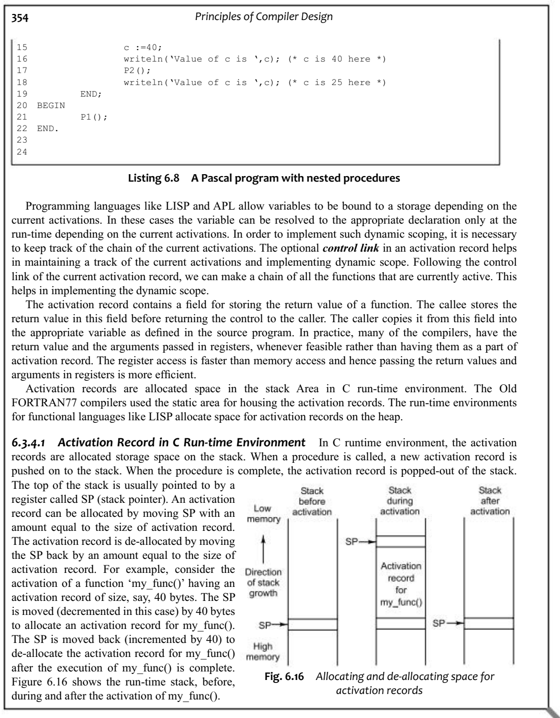
```
 1  PROGRAM sample(input,output);
 2  VAR p1,p2,p3 : integer;
 3
 4         FUNCTION my_func(f1,f2:integer): integer;
 5         BEGIN
 6               if (f1 > f2 )
 7               then
 8                      my_func := f1
 9               else
10                      my_func := f2;
11
12               f1 := 100 ;{ Changing the Value of Formal Parameter }
13               f2 := 120 ;{ Changing the Value of Formal Parameter }
14
15         END;
16
17  BEGIN
18
19         p1 := 4;
20         p2 := 30;
21
22         writeln('Before the function call p1=',p1,' p2=',p2);
23
24         p3 := my_func(p1,p2);
25
26         writeln('After the function call p1=',p1,' p2=',p2);
27
28  END.
```

Listing 6.12 *ex7.pas*

The dialog below shows the compilation and execution of the Pascal program shown in Listing 6.12 that uses the call-by-value mechanism for parameter-passing. The x86 assembly language output for the same program generated by the Pascal compiler—gpc is also seen in the dialog. We will use that to understand the details of implementing the call by value mechanism from a target code generator standpoint. Observing the execution of the program establishes the fact that any changes made to the parameters in a call-by-value method does not have any effect in the actual arguments at the caller site.

LISTINGS AND ALGORITHMS

Code Listings and Algorithm Specifications have been provided at appropriate locations in the chapters.



354 *Principles of Compiler Design*

```
                c :=40;
                writeln('Value of c is ',c); (* c is 40 here *)
                P2();
                writeln('Value of c is ',c); (* c is 25 here *)
        END;
BEGIN
        P1();
END.
```

Listing 6.8 A Pascal program with nested procedures

Programming languages like LISP and APL allow variables to be bound to a storage depending on the current activations. In these cases the variable can be resolved to the appropriate declaration only at the run-time depending on the current activations. In order to implement such dynamic scoping, it is necessary to keep track of the chain of the current activations. The optional ***control link*** in an activation record helps in maintaining a track of the current activations and implementing dynamic scope. Following the control link of the current activation record, we can make a chain of all the functions that are currently active. This helps in implementing the dynamic scope.

The activation record contains a field for storing the return value of a function. The callee stores the return value in this field before returning the control to the caller. The caller copies it from this field into the appropriate variable as defined in the source program. In practice, many of the compilers, have the return value and the arguments passed in registers, whenever feasible rather than having them as a part of activation record. The register access is faster than memory access and hence passing the return values and arguments in registers is more efficient.

Activation records are allocated space in the stack Area in C run-time environment. The Old FORTRAN77 compilers used the static area for housing the activation records. The run-time environments for functional languages like LISP allocate space for activation records on the heap.

6.3.4.1 ***Activation Record in C Run-time Environment*** In C runtime environment, the activation records are allocated storage space on the stack. When a procedure is called, a new activation record is pushed on to the stack. When the procedure is complete, the activation record is popped-out of the stack. The top of the stack is usually pointed to by a register called SP (stack pointer). An activation record can be allocated by moving SP with an amount equal to the size of activation record. The activation record is de-allocated by moving the SP back by an amount equal to the size of activation record. For example, consider the activation of a function 'my_func()' having an activation record of size, say, 40 bytes. The SP is moved (decremented in this case) by 40 bytes to allocate an activation record for my_func(). The SP is moved back (incremented by 40) to de-allocate the activation record for my_func() after the execution of my_func() is complete. Figure 6.16 shows the run-time stack, before, during and after the activation of my_func().

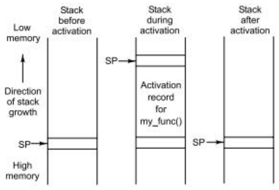
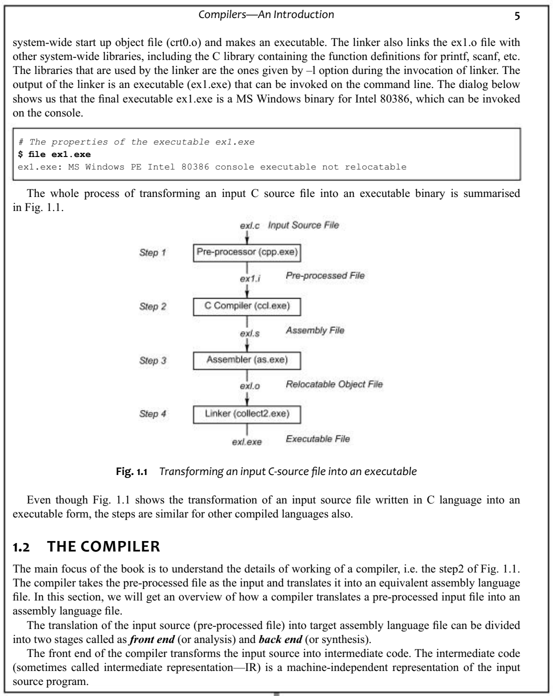


Fig. 6.16 *Allocating and de-allocating space for activation records*



Compilers—An Introduction 5

system-wide start up object file (crt0.o) and makes an executable. The linker also links the ex1.o file with other system-wide libraries, including the C library containing the function definitions for printf, scanf, etc. The libraries that are used by the linker are the ones given by –l option during the invocation of linker. The output of the linker is an executable (ex1.exe) that can be invoked on the command line. The dialog below shows us that the final executable ex1.exe is a MS Windows binary for Intel 80386, which can be invoked on the console.

```
# The properties of the executable ex1.exe
$ file ex1.exe
ex1.exe: MS Windows PE Intel 80386 console executable not relocatable
```

The whole process of transforming an input C source file into an executable binary is summarised in Fig. 1.1.

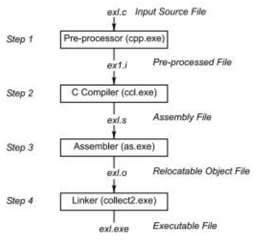


Fig. 1.1 *Transforming an input C-source file into an executable*

Even though Fig. 1.1 shows the transformation of an input source file written in C language into an executable form, the steps are similar for other compiled languages also.

1.2 THE COMPILER

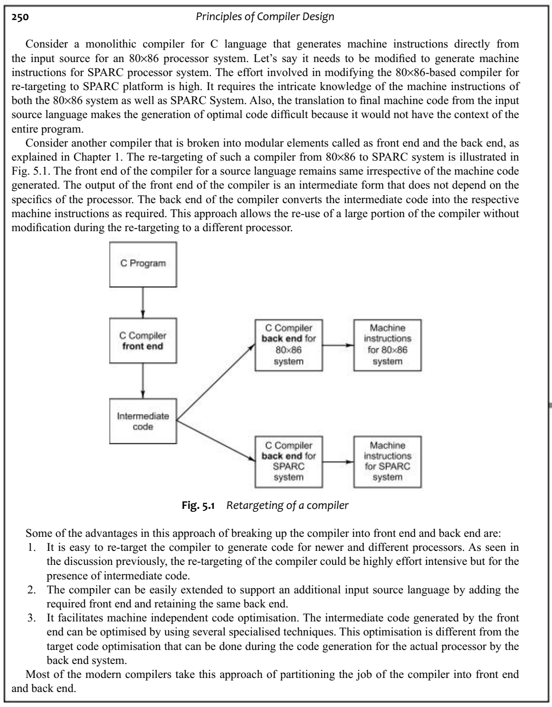
The main focus of the book is to understand the details of working of a compiler, i.e. the step2 of Fig. 1.1. The compiler takes the pre-processed file as the input and translates it into an equivalent assembly language file. In this section, we will get an overview of how a compiler translates a pre-processed input file into an assembly language file.

The translation of the input source (pre-processed file) into target assembly language file can be divided into two stages called as ***front end*** (or analysis) and ***back end*** (or synthesis).

The front end of the compiler transforms the input source into intermediate code. The intermediate code (sometimes called intermediate representation—IR) is a machine-independent representation of the input source program.

FIGURES

Figures are used exhaustively in the text to illustrate the concepts and methods described.



250 *Principles of Compiler Design*

Consider a monolithic compiler for C language that generates machine instructions directly from the input source for an 80×86 processor system. Let's say it needs to be modified to generate machine instructions for SPARC processor system. The effort involved in modifying the 80×86-based compiler for re-targeting to SPARC platform is high. It requires the intricate knowledge of the machine instructions of both the 80×86 system as well as SPARC System. Also, the translation to final machine code from the input source language makes the generation of optimal code difficult because it would not have the context of the entire program.

Consider another compiler that is broken into modular elements called as front end and the back end, as explained in Chapter 1. The re-targeting of such a compiler from 80×86 to SPARC system is illustrated in Fig. 5.1. The front end of the compiler for a source language remains same irrespective of the machine code generated. The output of the front end of the compiler is an intermediate form that does not depend on the specifics of the processor. The back end of the compiler converts the intermediate code into the respective machine instructions as required. This approach allows the re-use of a large portion of the compiler without modification during the re-targeting to a different processor.

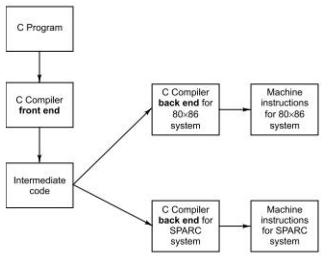
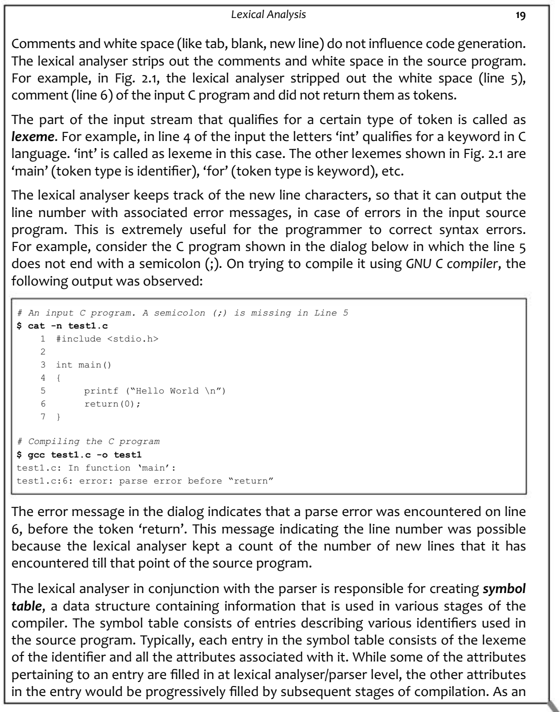

C Program
C Compiler front end
Intermediate code
C Compiler back end for 80×86 system
Machine instructions for 80×86 system
C Compiler back end for SPARC system
Machine instructions for SPARC system

Fig. 5.1 *Retargeting of a compiler*

Some of the advantages in this approach of breaking up the compiler into front end and back end are:

1. It is easy to re-target the compiler to generate code for newer and different processors. As seen in the discussion previously, the re-targeting of the compiler could be highly effort intensive but for the presence of intermediate code.
2. The compiler can be easily extended to support an additional input source language by adding the required front end and retaining the same back end.
3. It facilitates machine independent code optimisation. The intermediate code generated by the front end can be optimised by using several specialised techniques. This optimisation is different from the target code optimisation that can be done during the code generation for the actual processor by the back end system.

Most of the modern compilers take this approach of partitioning the job of the compiler into front end and back end.



Lexical Analysis 19

Comments and white space (like tab, blank, new line) do not influence code generation. The lexical analyser strips out the comments and white space in the source program. For example, in Fig. 2.1, the lexical analyser stripped out the white space (line 5), comment (line 6) of the input C program and did not return them as tokens.

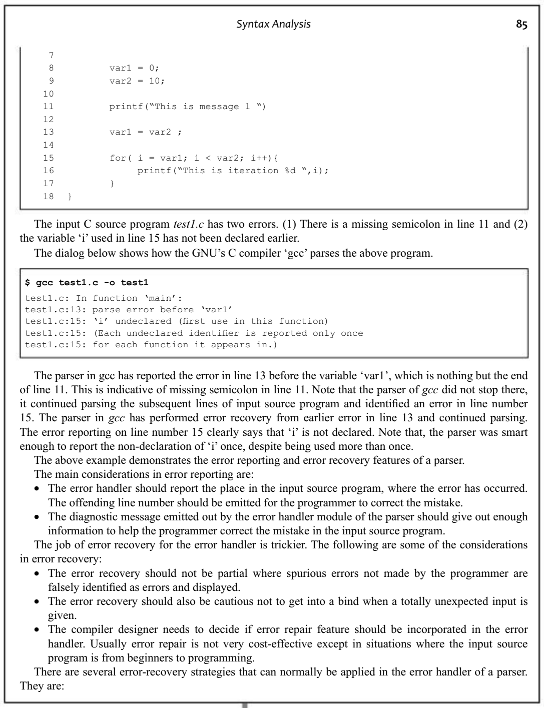
The part of the input stream that qualifies for a certain type of token is called as ***lexeme***. For example, in line 4 of the input the letters 'int' qualifies for a keyword in C language. 'int' is called as lexeme in this case. The other lexemes shown in Fig. 2.1 are 'main' (token type is identifier), 'for' (token type is keyword), etc.

The lexical analyser keeps track of the new line characters, so that it can output the line number with associated error messages, in case of errors in the input source program. This is extremely useful for the programmer to correct syntax errors. For example, consider the C program shown in the dialog below in which the line 5 does not end with a semicolon (;). On trying to compile it using *GNU C compiler*, the following output was observed:

```
# An input C program. A semicolon (;) is missing in Line 5
$ cat -n test1.c
    1  #include <stdio.h>
    2
    3  int main()
    4  {
    5       printf ("Hello World \n")
    6       return(0);
    7  }

# Compiling the C program
$ gcc test1.c -o test1
test1.c: In function 'main':
test1.c:6: error: parse error before "return"
```

The error message in the dialog indicates that a parse error was encountered on line 6, before the token 'return'. This message indicating the line number was possible because the lexical analyser kept a count of the number of new lines that it has encountered till that point of the source program.

The lexical analyser in conjunction with the parser is responsible for creating ***symbol table***, a data structure containing information that is used in various stages of the compiler. The symbol table consists of entries describing various identifiers used in the source program. Typically, each entry in the symbol table consists of the lexeme of the identifier and all the attributes associated with it. While some of the attributes pertaining to an entry are filled in at lexical analyser/parser level, the other attributes in the entry would be progressively filled by subsequent stages of compilation. As an



Syntax Analysis 85

```
    7
    8        var1 = 0;
    9        var2 = 10;
   10
   11        printf("This is message 1 ")
   12
   13        var1 = var2 ;
   14
   15        for( i = var1; i < var2; i++){
   16            printf("This is iteration %d ",i);
   17        }
   18  }
```

The input C source program *test1.c* has two errors. (1) There is a missing semicolon in line 11 and (2) the variable 'i' used in line 15 has not been declared earlier.

The dialog below shows how the GNU's C compiler 'gcc' parses the above program.

```
$ gcc test1.c -o test1
test1.c: In function 'main':
test1.c:13: parse error before 'var1'
test1.c:15: 'i' undeclared (first use in this function)
test1.c:15: (Each undeclared identifier is reported only once
test1.c:15: for each function it appears in.)
```

The parser in gcc has reported the error in line 13 before the variable 'var1', which is nothing but the end of line 11. This is indicative of missing semicolon in line 11. Note that the parser of *gcc* did not stop there, it continued parsing the subsequent lines of input source program and identified an error in line number 15. The parser in *gcc* has performed error recovery from earlier error in line 13 and continued parsing. The error reporting on line number 15 clearly says that 'i' is not declared. Note that, the parser was smart enough to report the non-declaration of 'i' once, despite being used more than once.

The above example demonstrates the error reporting and error recovery features of a parser.

The main considerations in error reporting are:

- The error handler should report the place in the input source program, where the error has occurred. The offending line number should be emitted for the programmer to correct the mistake.
- The diagnostic message emitted out by the error handler module of the parser should give out enough information to help the programmer correct the mistake in the input source program.

The job of error recovery for the error handler is trickier. The following are some of the considerations in error recovery:

- The error recovery should not be partial where spurious errors not made by the programmer are falsely identified as errors and displayed.
- The error recovery should also be cautious not to get into a bind when a totally unexpected input is given.
- The compiler designer needs to decide if error repair feature should be incorporated in the error handler. Usually error repair is not very cost-effective except in situations where the input source program is from beginners to programming.

There are several error-recovery strategies that can normally be applied in the error handler of a parser. They are:

Semantic Analysis 187

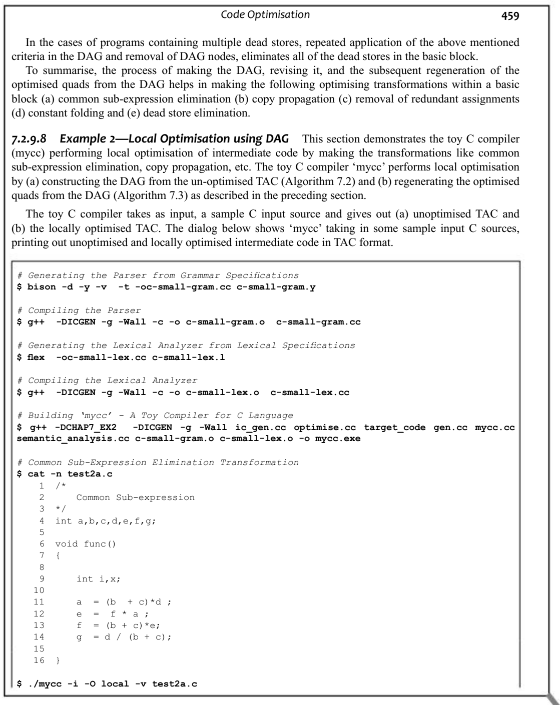
analysis is the last phase in which we reject incorrect input programs and flash error messages for the user to correct them.

The following dialog examines a few C programs, which have some semantic errors and shows us how the GNU C compiler detects and reports them. These examples give us a feel of what kinds of errors are detected in semantic analysis. Observe that all of these programs are syntactically correct, but have semantic errors.

```
# A C Program using an undeclared variable
$ cat -n sem_err1.c
    1
    2  int main()
    3  {
    4       int a,b;
    5
    6       a=1;
    7       b=2;
    8       c=3; /* Use of undeclared variable */
    9
   10       a = b + c;
   11
   12       return(a);
   13
   14  }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err1.c -o sem_err1
sem_err1.c: In function `main':
sem_err1.c:8: error: `c' undeclared (first use in this function)
sem_err1.c:8: error: (Each undeclared identifier is reported only once
sem_err1.c:8: error: for each function it appears in.)

# A C Program Assigning a float to char pointer
$ cat -n sem_err2.c
    1
    2  int main()
    3  {
    4       char *a;
    5
    6       float b,c;
    7
    8       b = 30.45;
    9       c = 40.36;
   10
   11       a = b + c; /* Assigning a float to char pointer */
   12
   13       return(0);
   14
   15  }
```

PRODUCTION COMPILER REFERENCES

Dialogs exemplifying the behaviour of a production compiler suite (gcc) have been provided in the pertinent sections of the textbook.



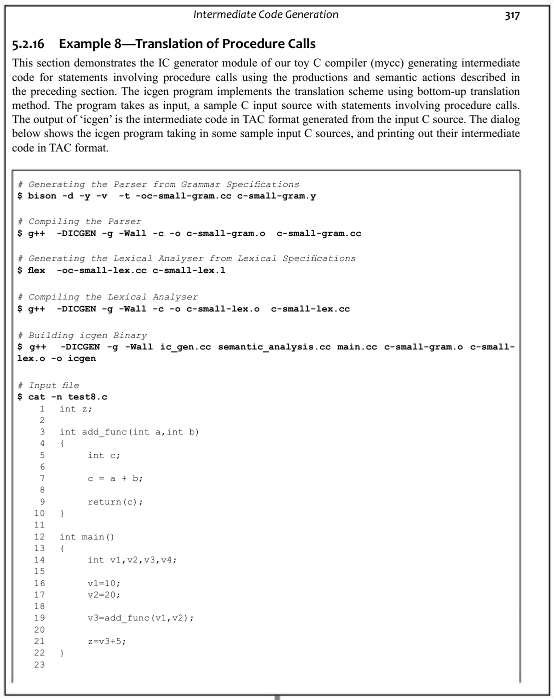
In the cases of programs containing multiple dead stores, repeated application of the above mentioned criteria in the DAG and removal of DAG nodes, eliminates all of the dead stores in the basic block.

To summarise, the process of making the DAG, revising it, and the subsequent regeneration of the optimised quads from the DAG helps in making the following optimising transformations within a basic block (a) common sub-expression elimination (b) copy propagation (c) removal of redundant assignments (d) constant folding and (e) dead store elimination.

7.2.9.8 ***Example 2—Local Optimisation using DAG*** This section demonstrates the toy C compiler (mycc) performing local optimisation of intermediate code by making the transformations like common sub-expression elimination, copy propagation, etc. The toy C compiler 'mycc' performs local optimisation by (a) constructing the DAG from the un-optimised TAC (Algorithm 7.2) and (b) regenerating the optimised quads from the DAG (Algorithm 7.3) as described in the preceding section.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC and (b) the locally optimised TAC. The dialog below shows 'mycc' taking in some sample input C sources, printing out unoptimised and locally optimised intermediate code in TAC format.

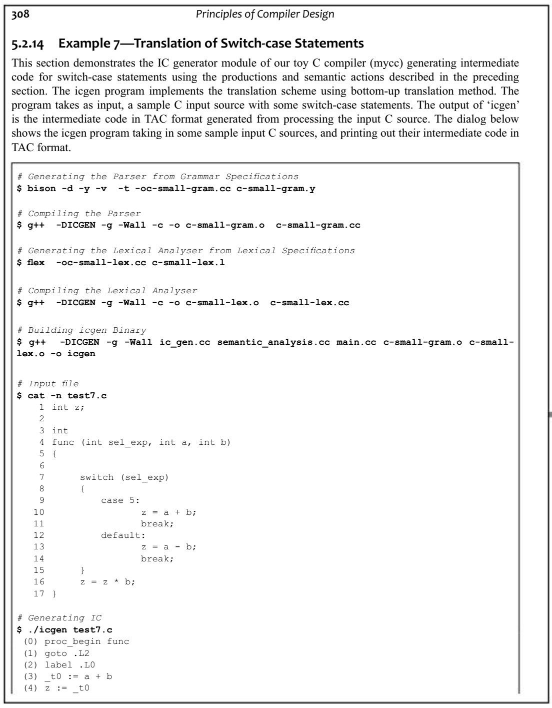
```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$ g++ -DCHAP7_EX2   -DICGEN -g -Wall ic_gen.cc optimise.cc target_code gen.cc mycc.cc
semantic_analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Common Sub-Expression Elimination Transformation
$ cat -n test2a.c
     1  /*
     2      Common Sub-expression
     3  */
     4  int a,b,c,d,e,f,g;
     5
     6  void func()
     7  {
     8
     9      int i,x;
    10
    11      a  = (b  + c)*d ;
    12      e  =  f * a ;
    13      f  = (b + c)*e;
    14      g  = d / (b + c);
    15
    16  }

$ ./mycc -i -O local -v test2a.c
```



5.2.16 Example 8—Translation of Procedure Calls

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving procedure calls using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with statements involving procedure calls. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++   -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test8.c
     1   int z;
     2
     3   int add_func(int a,int b)
     4   {
     5        int c;
     6
     7        c = a + b;
     8
     9        return(c);
    10   }
    11
    12   int main()
    13   {
    14        int v1,v2,v3,v4;
    15
    16        v1=10;
    17        v2=20;
    18
    19        v3=add_func(v1,v2);
    20
    21        z=v3+5;
    22   }
    23
```



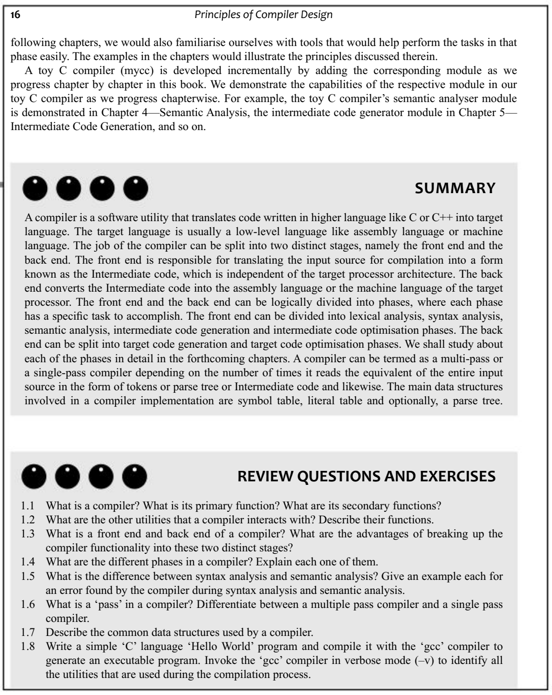
5.2.14 Example 7—Translation of Switch-case Statements

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for switch-case statements using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with some switch-case statements. The output of 'icgen' is the intermediate code in TAC format generated from processing the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++   -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test7.c
     1 int z;
     2
     3 int
     4 func (int sel_exp, int a, int b)
     5 {
     6
     7      switch (sel_exp)
     8      {
     9          case 5:
    10                  z = a + b;
    11                  break;
    12          default:
    13                  z = a - b;
    14                  break;
    15      }
    16      z = z * b;
    17 }

# Generating IC
$ ./icgen test7.c
 (0) proc_begin func
 (1) goto .L2
 (2) label .L0
 (3) _t0 := a + b
 (4) z := _t0
```

A TOY C COMPILER IMPLEMENTATION

A Toy C Language compiler is built progressively chapter by chapter using the concepts explained in each chapter.



16 Principles of Compiler Design

following chapters, we would also familiarise ourselves with tools that would help perform the tasks in that phase easily. The examples in the chapters would illustrate the principles discussed therein.

A toy C compiler (mycc) is developed incrementally by adding the corresponding module as we progress chapter by chapter in this book. We demonstrate the capabilities of the respective module in our toy C compiler as we progress chapterwise. For example, the toy C compiler's semantic analyser module is demonstrated in Chapter 4—Semantic Analysis, the intermediate code generator module in Chapter 5—Intermediate Code Generation, and so on.

SUMMARY

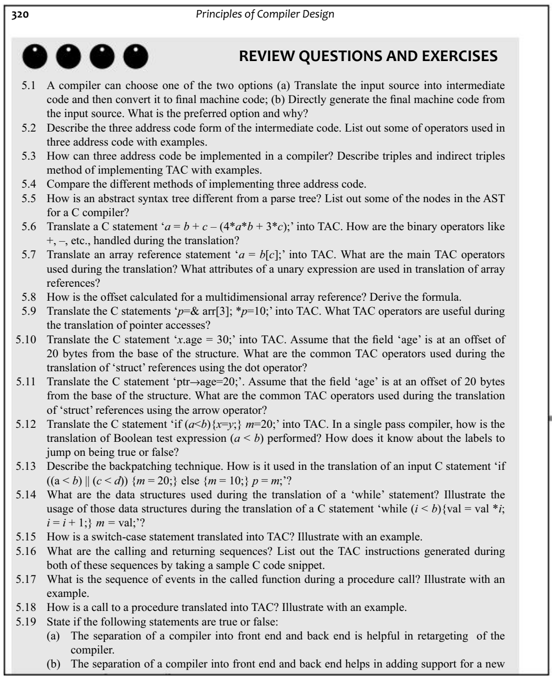
A compiler is a software utility that translates code written in higher language like C or C++ into target language. The target language is usually a low-level language like assembly language or machine language. The job of the compiler can be split into two distinct stages, namely the front end and the back end. The front end is responsible for translating the input source for compilation into a form known as the Intermediate code, which is independent of the target processor architecture. The back end converts the Intermediate code into the assembly language or the machine language of the target processor. The front end and the back end can be logically divided into phases, where each phase has a specific task to accomplish. The front end can be divided into lexical analysis, syntax analysis, semantic analysis, intermediate code generation and intermediate code optimisation phases. The back end can be split into target code generation and target code optimisation phases. We shall study about each of the phases in detail in the forthcoming chapters. A compiler can be termed as a multi-pass or a single-pass compiler depending on the number of times it reads the equivalent of the entire input source in the form of tokens or parse tree or Intermediate code and likewise. The main data structures involved in a compiler implementation are symbol table, literal table and optionally, a parse tree.

REVIEW QUESTIONS AND EXERCISES

1.1 What is a compiler? What is its primary function? What are its secondary functions?
1.2 What are the other utilities that a compiler interacts with? Describe their functions.
1.3 What is a front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages?
1.4 What are the different phases in a compiler? Explain each one of them.
1.5 What is the difference between syntax analysis and semantic analysis? Give an example each for an error found by the compiler during syntax analysis and semantic analysis.
1.6 What is a 'pass' in a compiler? Differentiate between a multiple pass compiler and a single pass compiler.
1.7 Describe the common data structures used by a compiler.
1.8 Write a simple 'C' language 'Hello World' program and compile it with the 'gcc' compiler to generate an executable program. Invoke the 'gcc' compiler in verbose mode (–v) to identify all the utilities that are used during the compilation process.

SUMMARY

Each chapter material is accompanied by a summary section that gives the reader a quick glimpse of what has been learnt in the chapter broadly.



320 Principles of Compiler Design

REVIEW QUESTIONS AND EXERCISES

5.1 A compiler can choose one of the two options (a) Translate the input source into intermediate code and then convert it to final machine code; (b) Directly generate the final machine code from the input source. What is the preferred option and why?
5.2 Describe the three address code form of the intermediate code. List out some of operators used in three address code with examples.
5.3 How can three address code be implemented in a compiler? Describe triples and indirect triples method of implementing TAC with examples.
5.4 Compare the different methods of implementing three address code.
5.5 How is an abstract syntax tree different from a parse tree? List out some of the nodes in the AST for a C compiler?
5.6 Translate a C statement '$a = b + c - (4*a*b + 3*c)$;' into TAC. How are the binary operators like +, –, etc., handled during the translation?
5.7 Translate an array reference statement '$a = b[c]$;' into TAC. What are the main TAC operators used during the translation? What attributes of a unary expression are used in translation of array references?
5.8 How is the offset calculated for a multidimensional array reference? Derive the formula.
5.9 Translate the C statements '*p*=& arr[3]; **p*=10;' into TAC. What TAC operators are useful during the translation of pointer accesses?
5.10 Translate the C statement '*x*.age = 30;' into TAC. Assume that the field 'age' is at an offset of 20 bytes from the base of the structure. What are the common TAC operators used during the translation of 'struct' references using the dot operator?
5.11 Translate the C statement 'ptr→age=20;'. Assume that the field 'age' is at an offset of 20 bytes from the base of the structure. What are the common TAC operators used during the translation of 'struct' references using the arrow operator?
5.12 Translate the C statement 'if (a<b){x=y;} m=20;' into TAC. In a single pass compiler, how is the translation of Boolean test expression ($a < b$) performed? How does it know about the labels to jump on being true or false?
5.13 Describe the backpatching technique. How is it used in the translation of an input C statement 'if (($a < b$) || ($c < d$)) {$m = 20$;} else {$m = 10$;} $p = m$;'?
5.14 What are the data structures used during the translation of a 'while' statement? Illustrate the usage of those data structures during the translation of a C statement 'while ($i < b$){val = val *i; $i = i + 1$;} m = val;'?
5.15 How is a switch-case statement translated into TAC? Illustrate with an example.
5.16 What are the calling and returning sequences? List out the TAC instructions generated during both of these sequences by taking a sample C code snippet.
5.17 What is the sequence of events in the called function during a procedure call? Illustrate with an example.
5.18 How is a call to a procedure translated into TAC? Illustrate with an example.
5.19 State if the following statements are true or false:
(a) The separation of a compiler into front end and back end is helpful in retargeting of the compiler.
(b) The separation of a compiler into front end and back end helps in adding support for a new

REVIEW QUESTIONS AND EXERCISES

Review Questions and Exercises provided at the end of each chapter help the readers reinforce their learning.

COMPILERS—AN INTRODUCTION

1

Introduction

A compiler is a software utility that translates code written in higher language like C or C++ into target language. The target language is usually a low-level language like assembly language or machine language. The translation from higher language to low-level language is the primary job of the compiler. However, there are other important secondary functions that the compilers provide for helping the programmers develop software. Compilers provide for reporting errors and warnings in the input higher-language source to help the programmer in correcting them. Compilers allow options to help debug the execution of the executable program generated by it. Compilers offer options to generate extra 'profiling code' to report the statistics on the time taken by specific functions in the input source at the run time. Today's compilers offer many other programmer-friendly features that help us develop software quickly and correctly, meeting all the specified requirements.

In this chapter, we study the process of transforming code written in a higher language like C into an executable form. We get an overview of how a compiler translates code written in a higher level language like C or C++ into target language. The later chapters develop on the ideas presented here to give a detailed understanding of the compiler.

1.1 THE BIGGER PICTURE

A compiler works in tandem with a few other utilities like preprocessor, assembler, linker and so on to produce binaries that can be executed. In this section we look at the process of generating an executable binary from a sample input program written in C and understand how these utilities fit in.

Let's start off with a sample C program (ex1.c) and see how we transform it into a binary that can be executed. The following dialog shows the sample C program (ex1.c) being transformed into an executable using GNU's compiler collection (*gcc*)—a freely available compiler collection.

```
# A Sample input C fle
$ cat -n ex1.c
    1
 2     #defne SUCCESS 0
    3
    4  /* The function prototype for printf found in stdio.h */
    5  extern int printf(const char *, ...);
    6
 7     int main()
 8     {
 9           printf("Hello World\n");
 10
   11        /* returning 0 to the Operating system */
 12          return(SUCCESS);
 13    }

# Creating an executable from sample input fle using the GNU C compiler system
$ gcc -Wall ex1.c -o ex1

# Invoking the executable
$ ./ex1
Hello World
```

From the above dialog, we can understand that the GNU compiler collection (gcc) binary is invoked with the input C source file as the argument for compilation. This outputs an executable ex1, which can be invoked on the command line to get the desired effect. The above dialog abstracts us from a lot of behind-the-scene activity that is involved in conversion of the input C file into executable.

Let's try out the compilation of the same C program (ex1.c), this time with extra options to 'gcc' in order to know all the utilities that get involved in the transformation of ex1.c to an executable binary. The following dialog shows the compilation of ex1.c with gcc using the extra options for getting a detailed account of the compilation.

```
# Compiling with verbose option and preserving the intermediate fles
$ gcc --save-temps --verbose ex1.c -o ex1
Reading specs from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/specs
gcc version 2.95.3-5 (cygwin special)
 /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/cpp0.exe -lang-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -
D_X86_=1 -D_X86_=1 -Asystem(winnt) -Acpu(i386) -Amachine(i386) -Di386 -D__i386 -D__i386__ -Di686 -
Dpentiumpro -D__i686 -D__i686__ -D__pentiumpro -D__pentiumpro__ -
D__stdcall=__attribute__((__stdcall__)) -D__cdecl=__attribute__((__cdecl__)) -
D_stdcall=__attribute__((__stdcall__)) -D_cdecl=__attribute__((__cdecl__)) -
D__declspec(x)=__attribute__((x)) -D__CYGWIN32__ -D__CYG WIN__ -Dunix -D__unix__ -D__unix -isystem/
usr/local/include -idirafter /usr/include -idirafter /usr/include/w32api ex1.c ex1.i
```

```
GNU CPP version 2.95.3-5 (cygwin special) (80386, BSD syntax)
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/include
 /usr/include
 /usr/include/w32api
End of search list.
The following default directories have been omitted from the search path:
End of omitted list.
 /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/cc1.exe ex1.i -quiet -dumpbase ex1.c -version -o ex1.s
GNU C version 2.95.3-5 (cygwin special) (i686-pc-cygwin) compiled by GNU C version 2.95.3-5
cygwin special).
 /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/../../../../i686-pc-cygwin/bin/as.exe -o ex1.o ex1.s
 /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/collect2. exe  -Bdynamic --dll-search-prefx=cyg -o
ex1.exe /usr/lib/crt0.o -L/usr/local/lib -L/usr/lib -L/usr/lib/w32api -L/usr/lib/gcc-lib/i686-pc        -
cygwin/2.95.3-5 ex1.o -lgcc -lcygwin -luser32 -lkernel32 -ladvapi32 -lshell32 -lgcc
```

You can see from the above verbose dialog that there are 4 programs (shaded in gray) that get invoked internally for transforming the input source ex1.c into an executable ex1.exe. They are:

1. The pre-processor program (cpp0.exe)
2. The compiler (cc1.exe)
3. The assembler (as.exe)
4. The linker (collect2.exe)

Each one of these invocations is shaded in gray in the above dialog.

The pre-processor program (cpp0.exe) is the first program to get invoked. The main functions of a preprocessor are (a) to substitute the macros (like #define SUCCESS in the above example) with the exact value; (b) strip comments in the input file before parsing and (c) include the header files given by the #include statements. All the lines in the input C file having a '#' sign as the first character are handled by the pre-processor. The output of the pre-processor program is given by the file with '.i' extension. Let's check out the pre-processed output file of the above example.

```
# The pre-processed output fle - ex1.i
$ cat -n ex1.i
     1 # 1 "ex1.c"
     2
     3
     4
     5
     6 extern int printf(const char *, ...);
     7
     8 int main()
     9 {
    10      printf("Hello World\n");
    11
    12
    13      return(0 );
    14 }
```

The reader can verify in the pre-processed output file that the comment lines have been stripped out. The return statement in line 13 now uses the actual value of 0 instead of SUCCESS.

The C compiler (cc1.exe) is the next program that gets invoked. This takes the pre-processed file (ex1.i) as input and creates an assembly language output stored in a file ex1.s. This is the compiler functionality that we are attempting to understand as the core of this book. The compiler utility translates code written in C programming language (after pre-processing) into target assembly language. The assembly language output file (ex1.s) created for the sample C program that we compiled is shown below.

```
# The assembly language output fle - ex1.s
$ cat -n ex1.s
     1       .fle "ex1.c"
     2 gcc2_compiled.:
     3 __gnu_compiled_c:
     4       .def ___main;    .scl 2;    .type 32;    .endef
     5 .text
     6 LC0:
     7       .ascii "Hello World\12\0"
     8       .align 4
     9 .globl _main
    10       .def    _main;    .scl  2;    .type 32;    .endef
    11 _main:
    12       pushl %ebp
    13       movl %esp,%ebp
    14       subl $8,%esp
    15       call ___main
    16       addl $-12,%esp
    17       pushl $LC0
    18       call _printf
    19       addl $16,%esp
    20       xorl %eax,%eax
    21       jmp L2
    22       .align 4
    23 L2:
    24       movl %ebp,%esp
    25       popl %ebp
    26       ret
    27       .def _printf; .scl 2; .type 32; .endef
```

The third step is the invocation of assembler (as.exe) to convert the assembly file ex1.s into a relocatable object file ex1.o. The relocatable object file is not in human readable form. There are several formats for the relocatable object file. Some of the common object file formats are ELF(Executable and Linking Format) and COFF (Common Object File Format). We can check out which format our sample program has been converted to by using a convenience utility called the 'file' to give out the characteristics of the object file. The dialog below tells us that the object file ex1.o is in COFF Format.

```
# Figuring out the object fle format
$ file ex1.o
ex1.o: 80386 COFF executable not stripped - version 30821
```

The fourth and final step is the invocation of the linker (collect2.exe) to generate the executable ex1.exe given the object file ex1.o as the input. The linker utility links the relocatable object file (ex1.o) with the system-wide start up object file (crt0.o) and makes an executable. The linker also links the ex1.o file with other system-wide libraries, including the C library containing the function definitions for printf, scanf, etc. The libraries that are used by the linker are the ones given by –l option during the invocation of linker. The output of the linker is an executable (ex1.exe) that can be invoked on the command line. The dialog below shows us that the final executable ex1.exe is a MS Windows binary for Intel 80386, which can be invoked on the console.

```
# The properties of the executable ex1.exe
$ file ex1.exe
ex1.exe: MS Windows PE Intel 80386 console executable not relocatable
```

The whole process of transforming an input C source file into an executable binary is summarised in Fig. 1.1.

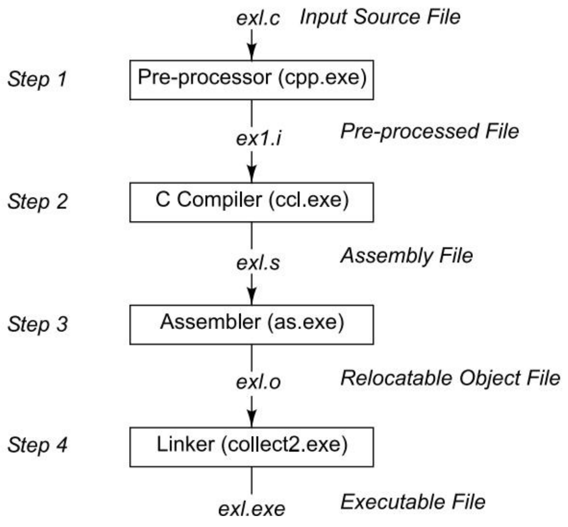


Fig. 1.1 *Transforming an input C-source file into an executable*

Even though Fig. 1.1 shows the transformation of an input source file written in C language into an executable form, the steps are similar for other compiled languages also.

1.2 THE COMPILER

The main focus of the book is to understand the details of working of a compiler, i.e. the step2 of Fig. 1.1. The compiler takes the pre-processed file as the input and translates it into an equivalent assembly language file. In this section, we will get an overview of how a compiler translates a pre-processed input file into an assembly language file.

The translation of the input source (pre-processed file) into target assembly language file can be divided into two stages called as ***front end*** (or analysis) and ***back end*** (or synthesis).

The front end of the compiler transforms the input source into intermediate code. The intermediate code (sometimes called intermediate representation—IR) is a machine-independent representation of the input source program.

The back end of the compiler takes the machine-independent intermediate code and generates the target assembly language program. The backend deals with machine-specific details like the registers, number of allowable operators, and so on.

Figure 1.2 illustrates the two-stage design approach of a compiler using C language source file as the input.

The main advantages of having this two-stage approach are:

- The compiler can be extended to support an additional processor by adding the required back end of the compiler. The existing front end is completely re-used in this case. This is illustrated in Fig. 1.3.
- The compiler can be easily extended to support an additional input source language by adding the required front end. In this case, the back end is completely re-used. This is illustrated in Fig. 1.4.

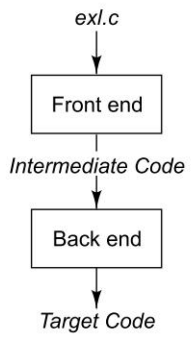


Fig. 1.2 *Front end and back end of the compiler*

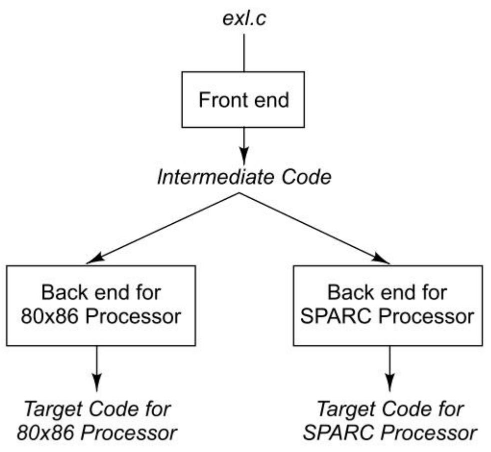

exl.c
Front end
Intermediate Code
Back end for 80x86 Processor
Back end for SPARC Processor
Target Code for 80x86 Processor
Target Code for SPARC Processor

Fig. 1.3 *Supporting an additional processor by adding back end*

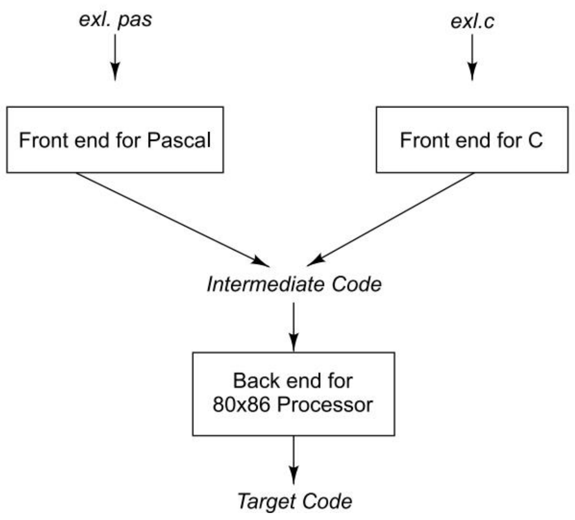


Fig. 1.4 *Supporting an additional language by adding front end*

1.2.1 Front End

The front end of the compiler is responsible for analysing the input source by breaking it into smaller entities, checking the syntax, verifying the semantics (meaning) and generating intermediate code. In this section, we learn about the front end of the compiler in more detail.

The front end is sub-divided into phases, where each phase is responsible for performing certain specific tasks. The front end of the compiler consists of the following phases (refer to Fig. 1.5).

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Intermediate code optimisation

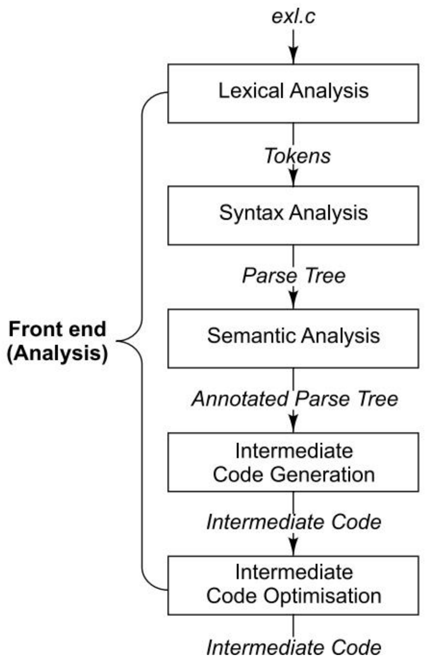
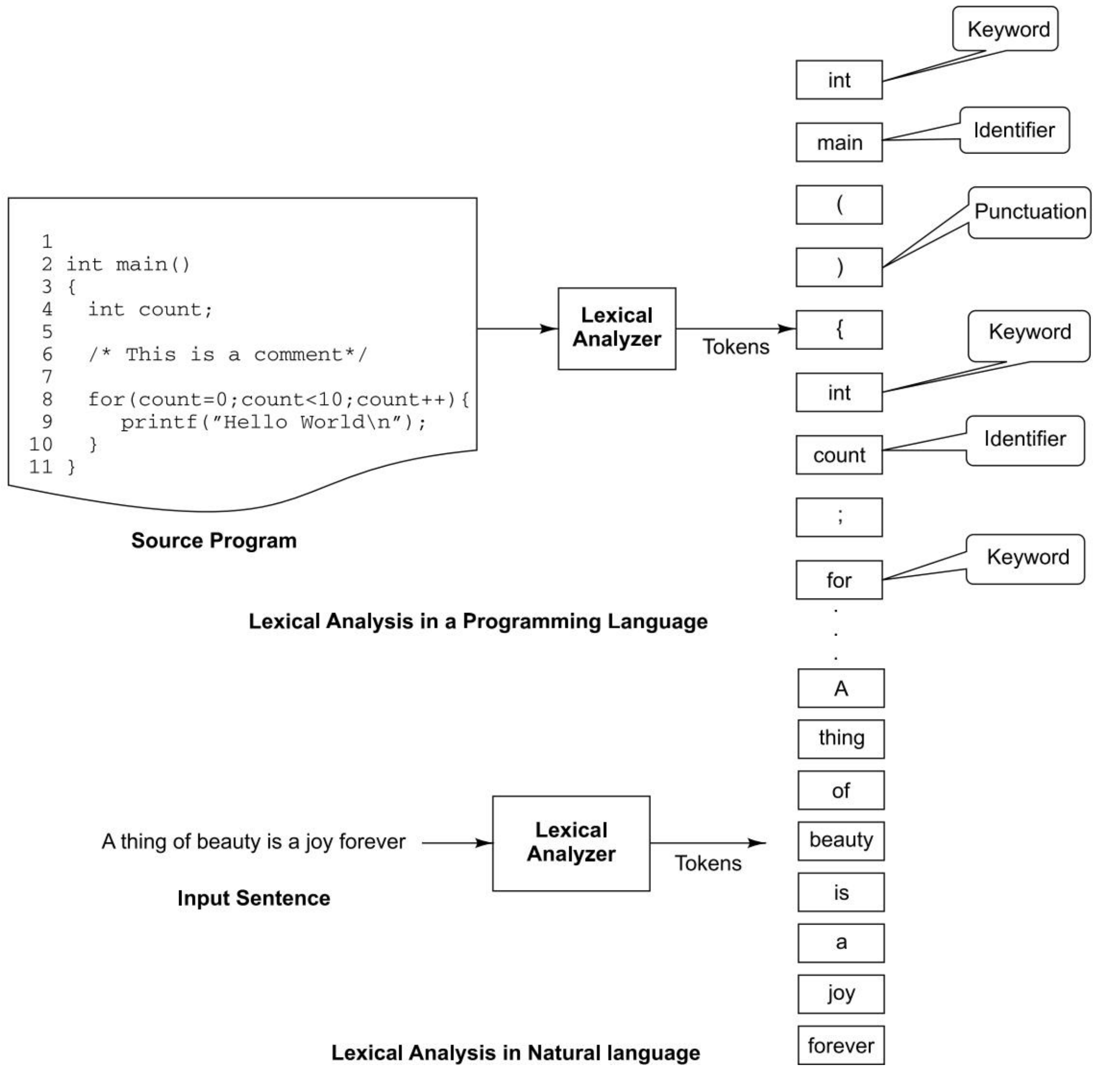


Fig. 1.5 *The phases in front end of compiler*

In ***lexical analysis***, the input source is broken up into a small meaningful sequence of characters called tokens. The tokens are the basic units of the programming language, which cannot be broken up further. Some examples of tokens in C programming language are Identifiers (user defined variables), keywords (like while, for), punctuation marks (like left brace/right brace, etc.) and operators (like + and –). Figure 1.6 shows how the input source is broken up into tokens during lexical analysis. The lexical analysis of the input source program can be compared to breaking up of a sentence into discrete words, which form the basic units in a natural language.

In ***syntax analysis***, the tokens are grouped together and checked, if they form a valid sequence as defined in the programming language. A ***context-free grammar*** specifies the rules or productions for identifying constructs that are valid in a programming language. The productions can be compared to the rules of grammar in natural languages.



```
int main()
{
  int count;

  /* This is a comment*/

  for(count=0;count<10;count++){
     printf("Hello World\n");
  }
}
```

Fig. 1.6 *The lexical analysis*

To get a feel of the context-free grammar and syntax analysis, let's consider a small set of grammar rules that can be used to recognise a few English sentences.

1. A sentence consists of a noun phrase followed by a verb phrase.
2. A noun phrase consists of an article followed by a noun.
3. An article can be the word 'a' or 'an' or 'the'.
4. A verb phrase consists of verb followed by noun phrase.
5. A noun can be the word 'boy' or 'bicycle'.
6. A verb can be the word 'rides'.

The above grammar rules can be written in context-free grammar as shown in Table 1.1. The rule 1 in Table 1.1 says that a sentence consists of a noun phrase followed by a verb phrase. The symbol ':=' is used to denote 'consists of'. The rule 3 in Table 1.1 tells us that an article can be 'a' or 'an' or 'the'. The symbol '|' is used to indicate a 'or' option.

Table 1.1 *Context-free grammar for recognising a few English sentences*

1	<sentence >	::=	<noun_phrase> <verb_phrase>
2	<noun_phrase>	::=	<article> <noun>
3	<article>	::=	a | an | the
4	<verb_phrase>	::=	<verb> <noun_phrase>
5	<noun>	::=	boy | bicycle
6	<verb>	::=	rides

The following are valid sentences conforming to the language described by the syntax rules in Table 1.1.

The boy rides a bicycle.
A boy rides a bicycle.
The boy rides the bicycle.

In a similar way for a programming language, a simple C assignment statement can be defined using the context-free grammar shown in Table 1.2. Production 1 states that a C statement consists of an identifier followed by an equal to ('=') sign, followed by a C expression followed by a semi-colon. Production 2 states that a C expression can be a constant. Production 3 states that C expression can be an identifier. Production 4 states that a C expression can be a C expression followed by a '+' operator followed by another C expression. Observe that the fourth production is recursive in nature. The tokens that are used in these rules are IDENTIFIER, CONSTANT, '=' , ';' and '+'.

Table 1.2 *Context-free grammar for a simple C assignment statement*

1	<c_statement >	::=	IDENTIFIER '=' <c_expression> ';'
2	<c_expression>	::=	CONSTANT
3		|	IDENTIFIER
4		|	<c_expression> '+' <c_expression>

The context-free grammar in Table 1.2 recognises the following C statements:

$x = y + 5;$
$x = y + z + 5;$

The outcome of syntax analysis can be a ***parse tree***. A parse tree is a record of which productions have been used to ascertain that the input is part of the language. Figure 1.7 shows visually the parse tree of an input C statement ($x = y + z$;) using the context-free grammar Table 1.2. The parse tree for an input English sentence (The boy rides a bicycle) using the context-free grammar in Table 1.1 is also shown in Fig. 1.7.

In ***semantic analysis***, we check if the syntactically correct statements make a meaningful reading. For example, a statement in the input source program '$x = y + 2$;' would not make a meaningful read if say x is the name of a function or array and y is a float type of variable. This statement might be syntactically acceptable by the productions of the context-free grammar in syntax analysis, but would not hold out during semantic analysis because the data types of x and y are not compatible. In natural language parlance, this is very similar to having a grammatically correct sentence, but devoid of meaning. For example, the syntax rules in Table 1.1 would also accept a sentence "The bicycle rides the boy". This sentence does not make sense, so we would reject it during semantic analysis. In a similar way, a C statement *'myfunc=y;'*, where *myfunc* is the name of a function and *y*, a float type might be acceptable in syntax analysis, but would be rejected in semantic analysis since the data types of left-hand side and right-hand side do not match. Most of the semantic analysis revolves around such type checking. Other tasks in semantic analysis involve detection of undeclared variables, access violations and so on. The result of semantic analysis is annotating the parse tree with more information on the data types.

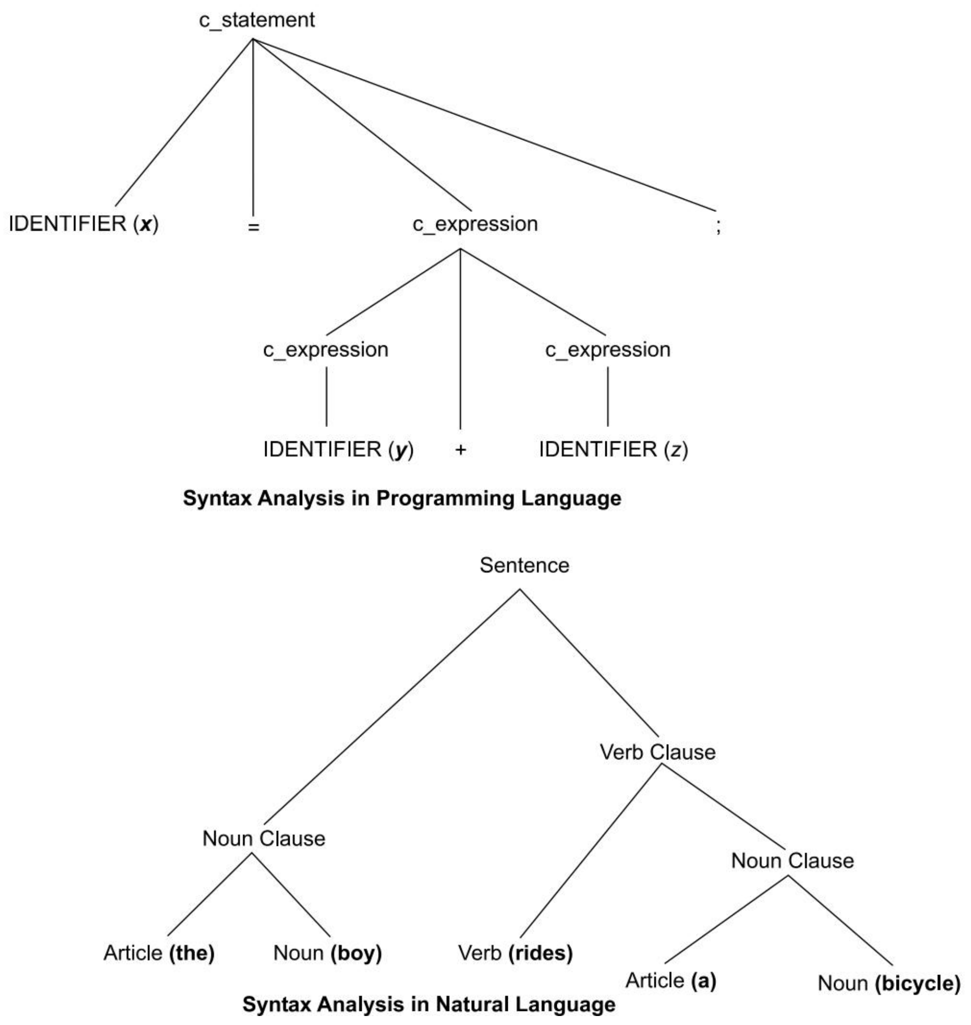


Fig. 1.7 *Syntax analysis*

In the ***intermediate code generation*** phase, we walk through the annotated parse tree and generate intermediate code. Three-address code (TAC) is one of the common forms of Intermediate code. Three-address code is a sequence of instructions, each of which can have at most three operands. Table 1.3 shows a sample input source and its equivalent Intermediate code. The variables *t2*, *t3*, etc. shown in the intermediate code are compiler generated temporary variables. We use the three-address code intermediate form extensively in this book. However, other forms of intermediate code like directed acyclic graphs (DAG), etc. would also be studied in later chapters.

Table 1.3 *Intermediate code*

Input C Statement	Intermediate Code

`v4 = v5 + 34 - (45 *v1 - v2);`	`t2 := v5 + 34`
	`t3 := 45 * v1`
	`t4 := t3 - v2`
	`t5 := t2 - t4`
	`v4 := t5`

In the ***intermediate code optimisation*** phase, the intermediate code is optimised by using several techniques, like elimination of common sub-expressions, elimination of unreachable code segments, elimination of statements that are not modified in the loop, and so on. Table 1.4 shows an Intermediate code segment before and after elimination of common sub-expression.

Table 1.4 *Intermediate code optimisation*

Input C Statement	Intermediate Code	Intermediate Code after elimination of Common sub-expression

`a = b + c * d ;`
`e = a + c * d ;`
	`t1: = c * d`
	`a : = b + t1`
	`t2: = c * d`
	`e : = a + t2`
		`t1 : = c * d`
		`a  : = b + t1`
		`e  : = a + t1`

The optimised Intermediate code emerging out of the Intermediate code optimisation phase is passed on to the back end of the compiler for generation of target code.

1.2.2 Back End

The back end of the compiler is responsible for translating the machine-independent intermediate code into the target assembly language. The back end of the compiler depends on the target processor, where the final binary would be executed. In this section, we study about the back end of the compiler in greater detail.

The back end of the compiler consists of the following phases (refer to Fig. 1.8).

- Target code generation
- Target code optimisation

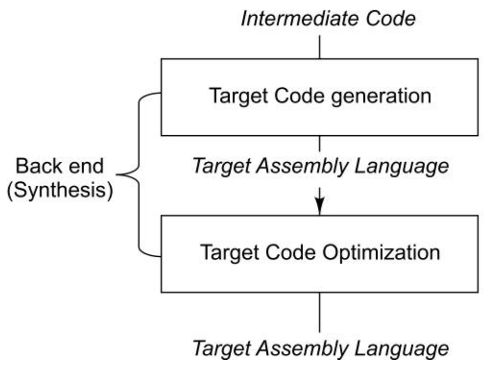


Fig. 1.8 *Phases in the back end of the compiler*

In ***target code generation*** phase, the intermediate code is translated into machine or assembly code.

In this phase, we associate memory locations with the variables and choose the appropriate assembly instructions depending on the target processor. The output of this phase is target code in the form of assembly language of the target hardware. Table 1.5 shows a segment of intermediate code and the generated target code in x86 assembly language (AT&T syntax) for the same.

Table 1.5 *Target code*

Intermediate Code	Target Code (x86 Assembly)

`x := y + z`	`movl _y, %eax`
	`addl _z,%eax`
	`movl %eax,_x`

In the ***target code optimisation*** phase the target code is transformed into a more efficient target code. The code optimisation phase considers better usage of registers, usage of machine-specific idioms or features of the processor like specialised instructions, pipelinings, and so on to make the generated target code more efficient. This phase also looks at replacing expensive operations like exponentiation by multiplication if possible. An example of target code in x86 assembly being optimised by using an auto-increment machine idiom is shown in Table 1.6.

Table 1.6 *Target code optimisation*

Intermediate Code	Target Code (x86 Assembly) before Code Optimization	Target Code (x86 Assembly) after Code Optimization

`x := x + 1`	`movl _x, %eax`
	`addl $1,%eax`
	`movl %eax,_x`
		`incl _x`

Table 1.7 gives the summary of the various phases and the tasks associated with them.

Table 1.7 *Compiler phases and the tasks*

#	Phase	Tasks
1	Lexical Analysis	Breaking up of the input source into tokens. Some examples of tokens are identifiers, constant, strings, punctuation marks (like ';'), operators. Adding identifiers into symbol table. Adding strings (like "Hello World") and constants into literal table.
2	Syntax Analysis	Group the tokens to see if they form a valid sequence as defined in the language. Add data-type Information to the symbol table entry.
3	Semantic Analysis	Perform data type checks to determine, if the data types of the operands are compatible and report errors for incompatible ones.
4	Intermediate Code Generation	Generate the intermediate code. The intermediate code is independent of the target machine.
5	Intermediate Code Optimisation	The generated intermediate code is optimised by eliminating common sub-expressions, unreachable code segments, statements that are not modified in the loop, and so on.
6	Target Code Generation	Generation of target code (assembly language) from the intermediate code. Associating memory locations with the variables.
7	Target Code Optimisation	Optimise the target code by using machine-specific idioms or features of the processor like specialised instructions, pipe lining, and so on. Replacing expensive operations like exponentiation by multiplication, if possible.

1.3 COMPILER IMPLEMENTATION

In this section, we consider ways of implementing a compiler based on 2-stage analysis-synthesis model as illustrated in Fig. 1.9, consolidating the phases in the front end and back end of the compiler. This section also gives us familiarity with some of the terminology used in the compiler implementations.

A simple and modular way of implementing a compiler based on the model shown in Fig. 1.9 is to have each phase as a module, with the input and output as specified in the figure itself. The lexical analysis module would scan the entire input source program, break it up into tokens and output the entire list of tokens. The syntax analysis module would take the entire list of tokens and create a parse tree that is representative of the entire source program. Next, the semantic analysis module would take the parse tree, perform type checking and annotate it with data-type information. In this manner, we can have modules for intermediate code generation, optimisation, and so on, where the module transforms the input and gives an output in accordance with Fig. 1.9. In this implementation, each of the modules runs through an input representative of the entire source program in some form (like tokens or parse tree or Intermediate code, etc.) and creates an output representative of the entire source program in same or different form. Each time we read through a representation of the entire program, we term it as a ***pass***. This type of implementation, where we read through a representation of entire source program multiple times is called as ***multi-pass compiler***. Going by this approach of having a pass for each of the phases in Fig. 1.9, we would have a 7-pass compiler implementation.

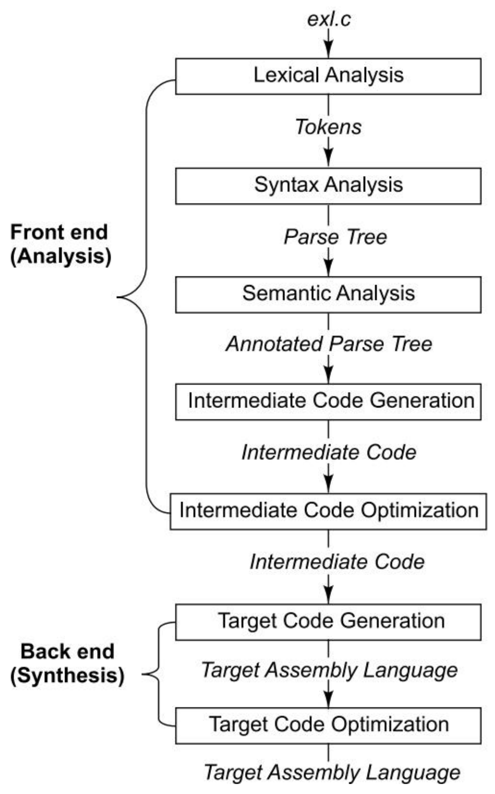


Fig. 1.9 *The front end and the back end of compiler*

Another way of implementing the compiler based on the model shown in Fig. 1.9 is to club several phases into one pass. For example, let's say we

1. Read a single token [lexical analysis].
2. If the token sequence matches a grammar rule go to step 3, else go to step 1 [syntax analysis].
3. Perform the semantic check for the matched grammar construct [semantic analysis].
4. Generate intermediate code for the matched grammar construct [intermediate code generation]

We can keep on repeating the sequence 1 through 4 above until the entire input source program is completely read. In this case the lexical analysis, syntax analysis, semantic analysis and Intermediate code generation are all performed in one single pass. Observe the control alternating among these phases in this implementation. The interface between each of these phases would be an implementation choice. However, the output of the complete pass remains as intermediate code in this case. Figure 1.10 shows a 4-pass compiler design, where we have clubbed the lexical analysis, syntax analysis, semantic analysis and intermediate code generation into one single pass and have a pass each for the rest of the phases. Typically, optimisation phases requires several passes because it involves looking at multiple instructions at one time to decide on a optimising method. In theory, if we eliminate optimisation, it is possible to have a single pass compiler for some of the programming languages like C.

Multi-pass compilation requires more memory since we need to store the output of each phase in totality. Multi-pass compiler also takes a longer time to compile, since it involves reading of the input in different forms (tokens, parse tree, etc.) multiple number of times.

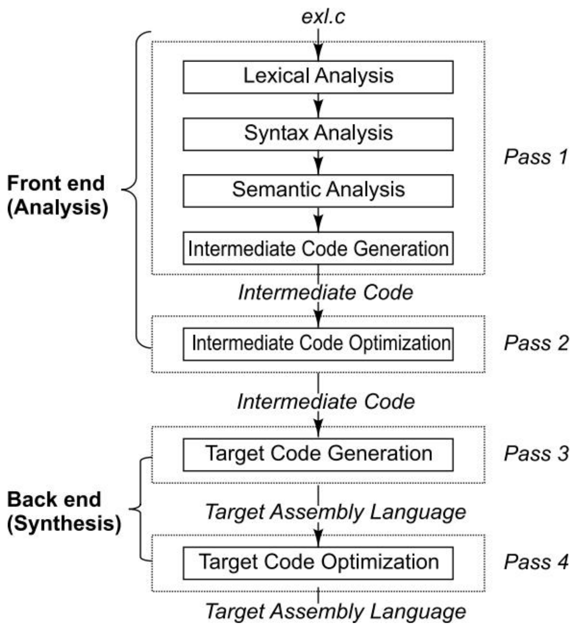


Fig. 1.10 *A four-pass compiler*

In practice, compilers are designed with the idea of keeping the number of passes to as minimum as possible. The number of passes required in a compiler to process an input source program depends on the structure of the programming language. Compilers for some of the programming languages like C can be implemented in a single pass, while a PL/1 or ALGOL 68 compiler cannot be implemented in a single pass. The programming languages PL/1 or ALGOL 68 allow for variables to be used before declaring them. This makes it hard to design a single pass compiler for them, since semantic analysis cannot be performed without the knowledge of the variable type.

1.4 DATA STRUCTURES IN A COMPILER

The various phases of compilers interact using some common data structures. There are two important tables that are used by various phases during the compilation. They are:

1. Symbol table
2. Literal table

A symbol table contains information with regard to the identifiers used in the input source program. Each entry in the symbol table corresponds to a symbol (Identifier) and contains details like the name of the symbol, the data type, size (the amount of memory required), and so on. Typically, an entry is made into the symbol table at the lexical analysis phase. The entry is updated and looked up in different phases of the compiler. In the syntax and semantic analysis phases, the entry is updated for details like the size and data type of the variable. During the target code generation phase, the size of the variable is used for generating appropriate target assembly code. In some compilers, there are symbol tables for every function along with a global symbol table (as shown in Fig. 1.11). These symbol tables might be maintained in a list or a stack. Another common scheme is to have a single symbol table for all the identifiers in the program with scope information present in the entry. The symbol table is accessed very frequently during the compilation of a program. A hash table is usually used for implementing a symbol table, mainly due to fast look-up capability.

Global Symbol Table

Name	Type	Size	Offset
fact5	INT	4	0
factorial	FUNC		
main	FUNC		

Local Symbol Table for Factorial ()

Name	Type	Size	Offset
n	INT	4	0
val	INT	4	4

```
int fact5;
int factorial (int n)
{
   int val;
   if (n>1){
     val=n*factorial (n-1);
     return (val);
   }else{
      return(1);
   }
}
int main ()
{
    print ("factorial program\n");
    fact5=factorial (5);
    printf ("fact5=%d \n",fact5);
}
```

Literal Table

Name	Type	Value
lit1	STRING	"Factorial Program\n"
lit2	CONST	5
lit3	STRING	"fact5=%d \n"

Fig. 1.11 *Symbol table and literal table for a sample input source*

A sample input C source and its corresponding symbol tables created in a compiler are shown in Fig. 1.11. Observe that each entry in the symbol table contains details like the name and the size of the variable (memory occupied) along with some other information that would be used in different phases of the compiler. The information shown in each entry of the symbol table in Fig. 1.11 is very preliminary. In practice there is much more information filled in for each entry.

A **literal table** stores the strings and constants found in the input source program. The main function of the literal table is to conserve memory by re-using the constants and the strings. In literal table, the information is usually entered at the time of lexical analysis and looked up during the target code generation. A sample input C source and its corresponding literal table created in a compiler are shown in Fig. 1.11.

The intermediate code is passed between the front end and the back end of the compiler. The intermediate code is also optimised during the optimisation phase. It is usually stored in an array/linked list of structures for facilitating easy reorganisation.

In compilers using a pass each for lexical, syntax and semantic analysis, the parse tree is another important data structure under consideration. It is usually implemented using a pointer-based structure, where the parent and children contain pointers to each other for facilitating quick traversal.

1.5 STUDY PLAN

This book is organised in accordance with the various phases of the compiler shown in Table 1.7, with a chapter being devoted to each one of the phases. During the discussion on the individual phases in the following chapters, we would also familiarise ourselves with tools that would help perform the tasks in that phase easily. The examples in the chapters would illustrate the principles discussed therein.

A toy C compiler (mycc) is developed incrementally by adding the corresponding module as we progress chapter by chapter in this book. We demonstrate the capabilities of the respective module in our toy C compiler as we progress chapterwise. For example, the toy C compiler's semantic analyser module is demonstrated in Chapter 4—Semantic Analysis, the intermediate code generator module in Chapter 5—Intermediate Code Generation, and so on.

SUMMARY

A compiler is a software utility that translates code written in higher language like C or C++ into target language. The target language is usually a low-level language like assembly language or machine language. The job of the compiler can be split into two distinct stages, namely the front end and the back end. The front end is responsible for translating the input source for compilation into a form known as the Intermediate code, which is independent of the target processor architecture. The back end converts the Intermediate code into the assembly language or the machine language of the target processor. The front end and the back end can be logically divided into phases, where each phase has a specific task to accomplish. The front end can be divided into lexical analysis, syntax analysis, semantic analysis, intermediate code generation and intermediate code optimisation phases. The back end can be split into target code generation and target code optimisation phases. We shall study about each of the phases in detail in the forthcoming chapters. A compiler can be termed as a multi-pass or a single-pass compiler depending on the number of times it reads the equivalent of the entire input source in the form of tokens or parse tree or Intermediate code and likewise. The main data structures involved in a compiler implementation are symbol table, literal table and optionally, a parse tree.

REVIEW QUESTIONS AND EXERCISES

1.1 What is a compiler? What is its primary function? What are its secondary functions?
1.2 What are the other utilities that a compiler interacts with? Describe their functions.
1.3 What is a front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages?
1.4 What are the different phases in a compiler? Explain each one of them.
1.5 What is the difference between syntax analysis and semantic analysis? Give an example each for an error found by the compiler during syntax analysis and semantic analysis.
1.6 What is a 'pass' in a compiler? Differentiate between a multiple pass compiler and a single pass compiler.
1.7 Describe the common data structures used by a compiler.
1.8 Write a simple 'C' language 'Hello World' program and compile it with the 'gcc' compiler to generate an executable program. Invoke the 'gcc' compiler in verbose mode (–v) to identify all the utilities that are used during the compilation process.

1.9 Compile a simple ‘C’ language ‘Hello World’ source program using gcc with --save-temps option to save the intermediate files after each step in the compilation process. Identify the output file after each of the steps, namely the pre-processing, compiling, assembling and linking. Use the ‘file’ utility to know more about each of the output files.

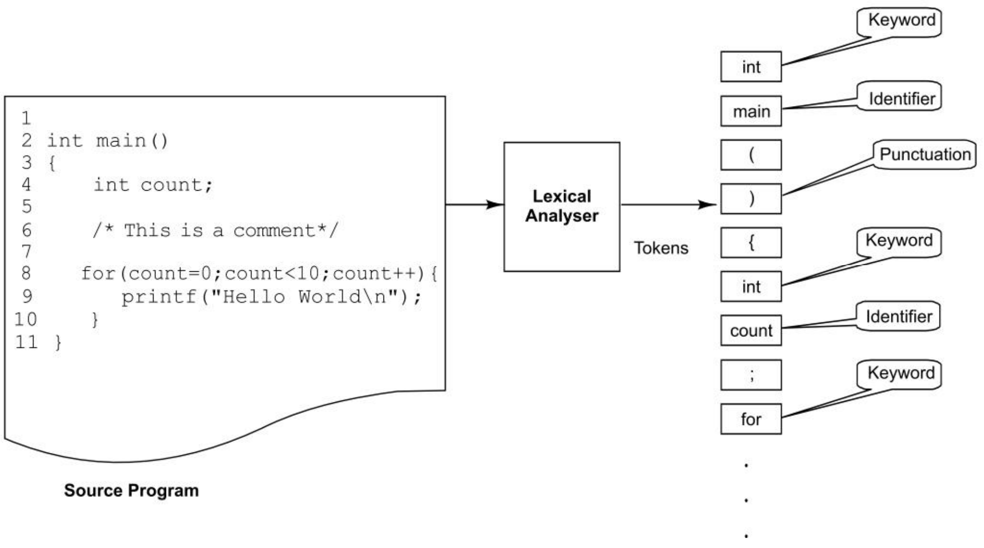
1.10 Does the ‘gcc’ compiler collection follow the analysis-synthesis model during compiler development? Are the benefits of using analysis-synthesis model obvious?

LEXICAL ANALYSIS

2

Introduction

Lexical analysis is the first stage in the compilation of a source program written in higher-level language like C or C++. The lexical analyser reads the input source program and produces as output, a sequence of tokens that the parser uses for syntax analysis. Consider for example a C program as input to the lexical analyser. The lexical analyser separates the input C program into various types of tokens like keywords, identifiers, operators, and so on as shown in Fig. 2.1.



```

int main()
{
    int count;

    /* This is a comment*/

   for(count=0;count<10;count++){
      printf("Hello World\n");
    }
}
```

Fig. 2.1 *Lexical analysis*

Comments and white space (like tab, blank, new line) do not influence code generation. The lexical analyser strips out the comments and white space in the source program. For example, in Fig. 2.1, the lexical analyser stripped out the white space (line 5), comment (line 6) of the input C program and did not return them as tokens.

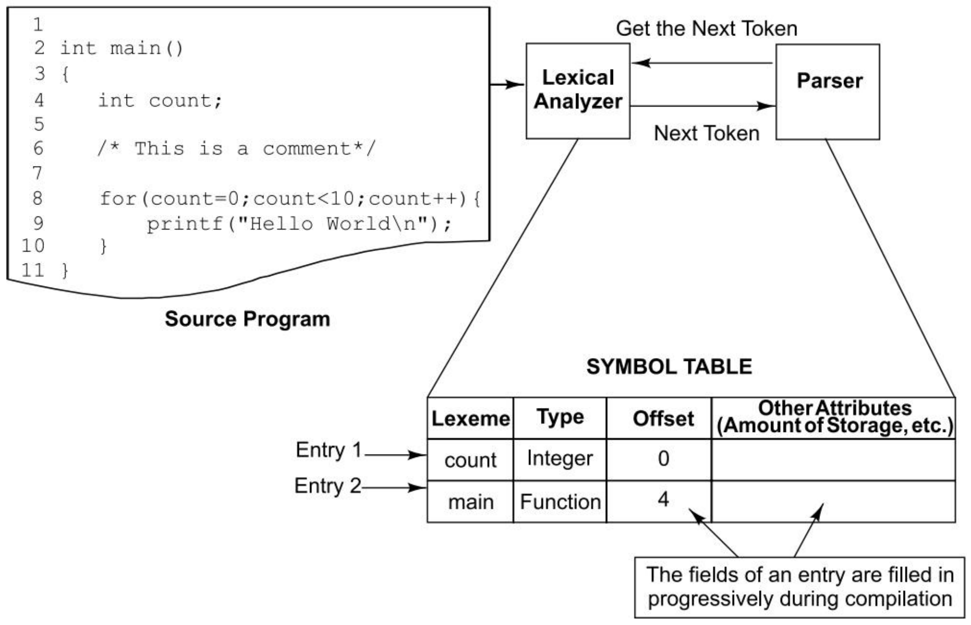
The part of the input stream that qualifies for a certain type of token is called as ***lexeme***. For example, in line 4 of the input the letters ‘int’ qualifies for a keyword in C language. ‘int’ is called as lexeme in this case. The other lexemes shown in Fig. 2.1 are ‘main’ (token type is identifier), ‘for’ (token type is keyword), etc.

The lexical analyser keeps track of the new line characters, so that it can output the line number with associated error messages, in case of errors in the input source program. This is extremely useful for the programmer to correct syntax errors. For example, consider the C program shown in the dialog below in which the line 5 does not end with a semicolon (;). On trying to compile it using *GNU C compiler*, the following output was observed:

```
# An input C program. A semicolon (;) is missing in Line 5
$ cat -n test1.c
     1  #include <stdio.h>
     2
     3  int main()
     4  {
     5       printf (“Hello World \n”)
     6       return(0);
     7  }

# Compiling the C program
$ gcc test1.c -o test1
test1.c: In function ‘main’:
test1.c:6: error: parse error before “return”
```

The error message in the dialog indicates that a parse error was encountered on line 6, before the token ‘return’. This message indicating the line number was possible because the lexical analyser kept a count of the number of new lines that it has encountered till that point of the source program.

The lexical analyser in conjunction with the parser is responsible for creating ***symbol table***, a data structure containing information that is used in various stages of the compiler. The symbol table consists of entries describing various identifiers used in the source program. Typically, each entry in the symbol table consists of the lexeme of the identifier and all the attributes associated with it. While some of the attributes pertaining to an entry are filled in at lexical analyser/parser level, the other attributes in the entry would be progressively filled by subsequent stages of compilation. As an example, consider a C program shown in Fig. 2.2. The lexical analyser in tandem with the parser would make an entry in the symbol table, indicating that the lexeme *main* is an identifier. The subsequent stages in compilation could add more information with respect to the entry, like the number of bytes of storage required for it, the position in the memory layout for the program, and so on. The symbol table is typically stored as a hash indexed on the lexeme.



```

int main()
{
   int count;

   /* This is a comment*/

   for(count=0;count<10;count++){
       printf("Hello World\n");
   }
}
```

Lexeme	Type	Offset	Other Attributes (Amount of Storage, etc.)
count	Integer	0	
main	Function	4	

Fig. 2.2 *Creation of symbol table*

2.1 ELEMENTS OF LEXICAL ANALYSIS

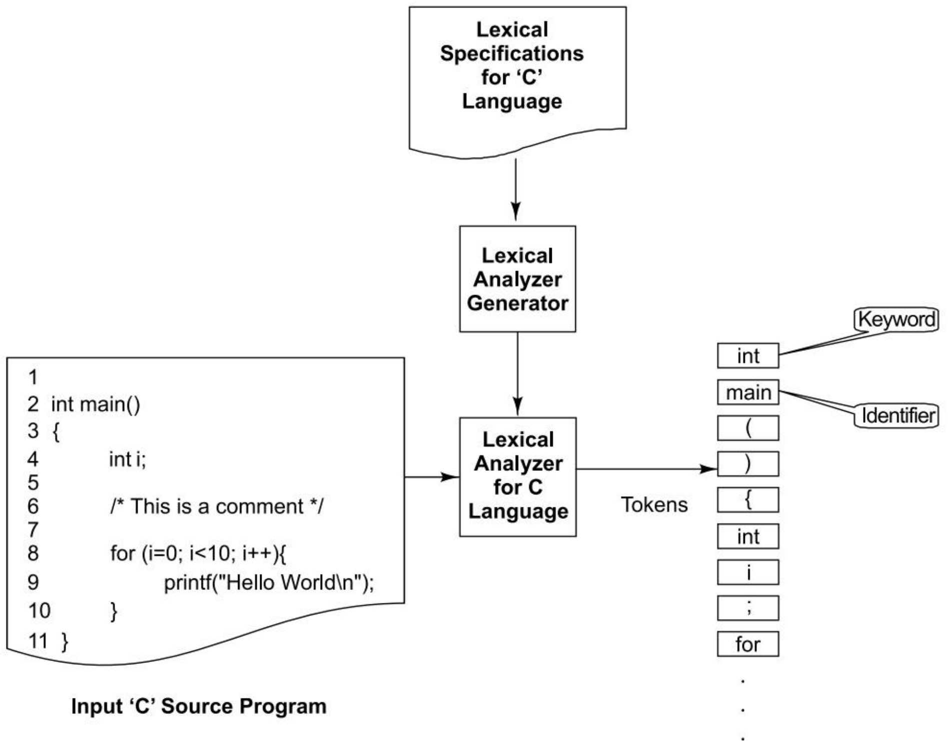
In the last section, we discussed the tasks that the lexical analyser does. We understood that the primary task of a lexical analyser is to break up the input source program into a sequence of tokens like identifier, keyword, string literals, constants, and so on. In this section, we discuss how a lexical analyser is made.

A lexical analyser for a specific programming language can be constructed by taking the input character-by-character and then checking in for various constructs of that language. For example, a lexical analyser for C language can take the input character-by-character and check if it is a keyword (like 'int' or 'char' or 'switch' or 'break', etc.) or operator or identifier or string literal, etc. Similarly a pascal lexical analyser can take in character-by-character and check if the input is a keyword (like 'FORMAT' or 'READ', etc.) or operator or identifier or string literal, etc. The difference between the C lexical analyser and the pascal lexical analyser is that the rules that define how to identify keyword, operator, identifier or a string literal will vary. For example, in C language, the keywords are 'switch', 'case', 'int', etc., while in pascal the keywords are 'record', 'var', 'then,' etc.

There are several difficulties with the above-mentioned approach of having the knowledge of the language tightly coupled with the lexical analyser:

- A lot of design/coding effort goes into parsing of the input that could be common to lexical analysers of any programming language.
- The complexity of the lexical analyser would be very high and adding a new construct to an existing language could become difficult.
- Developing a lexical analyser for a new language would be cumbersome and involve almost the same effort as any of the ones previously developed.

In order to overcome the difficulties mentioned above and facilitate the development of lexical analysers for any language easily, ***lexical analyser generators*** are used. A lexical analyser generator is a tool that can generate a code to perform lexical analysis of the input, given the rules for the basic building blocks of the language. The rules for the basic building blocks of a language are called its ***lexical specifications***. An example will make the terms clear. Assume that we are interested in developing lexical analyser for C language that can break up a C language program into tokens. In order to develop such a lexical analyser, we need to supply the lexical specifications for C language to a lexical analyser generator. The lexical analyser generator then transforms the lexical specifications into a lexical analyser that can be used to tokenize an input C program. This is illustrated in Fig. 2.3.



```

int main()
{
        int i;

        /* This is a comment */

        for (i=0; i<10; i++){
                printf("Hello World\n");
        }
}
```

Lexical Specifications for 'C' Language
Lexical Analyzer Generator
Lexical Analyzer for C Language
Tokens
Input 'C' Source Program
int
main
(
)
{
int
i
;
for
Keyword
Identifier

Fig. 2.3 *Lexical specifications, lexical analyser generator and lexical analyser*

The lexical specifications for any programming language consists of information about identifying each and every token that is defined for it. For example, the lexical specifications for C language would typically contain information about identifying keywords ('for', 'switch' 'case', etc.), operators (<, >, = etc.), identifier (starts with alphabet can have digits within it), string literal (within quotes), etc.

[illegible]

There are 3 steps to create a lexical analyser using *flex*.

1. Specify the tokens and their associated actions in a lexical specification file usually with a **'.l'** extension.
2. Compile the lexical specifications given in step 1 above using *flex* to generate a C file *lex.yy.c*.
3. Compile *lex.yy.c* using a C compiler to generate a binary file, which is the lexical analyser that transforms an input stream into a sequence of tokens.

This is shown in Fig. 2.4.

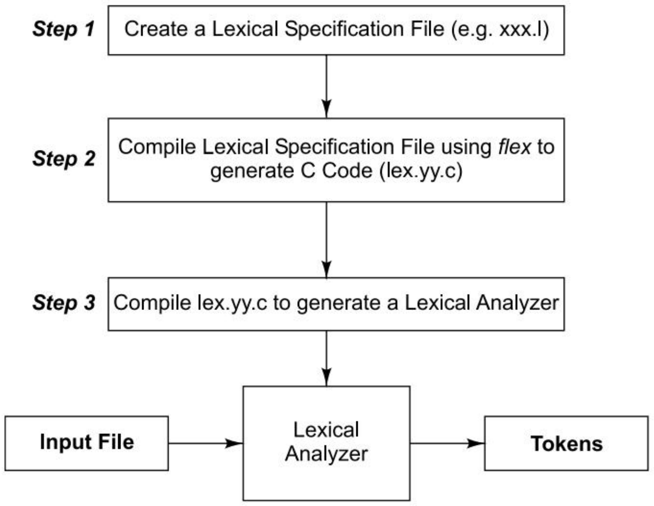

Step 1
Create a Lexical Specification File (e.g. xxx.l)
Step 2
Compile Lexical Specification File using flex to generate C Code (lex.yy.c)
Step 3
Compile lex.yy.c to generate a Lexical Analyzer
Input File
Lexical Analyzer
Tokens

Fig. 2.4 *Lexical analyser generation using flex*

To make the 3-step procedure clear, let's create a lexical analyser to split an English sentence into individual words. This lexical analyser will take an input file containing English sentences and tokenises them into individual words.

The lexical specification file to split an English sentence into individual words is '*words.l*' shown in Listing 2.1. The contents and format of the lexical specification file '*words.l*' will form the topic of discussion for the next section. For the illustration of the 3-step procedure, it suffices to assume that '*words.l*' file has already been created. The following dialog illustrates the 3-step procedure outlined in Fig. 2.4.

```
# The creation of words.l file is Step 1 . It is not shown here
# Compiling the lexical specification file, Step 2
$ flex words.l

# lex.yy.c generated by previous command
# Compiling it to build lexical analyzer, Step 3
$ gcc lex.yy.c -o eng_lex_analyzer -lfl

# Input to lexical analyzer
$ cat -n sentence
      1 The Quick Brown Fox jumps over a Lazy Dog
```

```
# lexical Analyzer at work on the input file
$ ./eng_lex_analyzer sentence
Lexeme=[The]    length=3  Token is WORD
Lexeme=[Quick]      length=5      Token is  WORD
Lexeme=[Brown]      length=5      Token is  WORD
Lexeme=[Fox]    length=3  Token is  WORD
Lexeme=[jumps]      length=5  Token is  WORD
Lexeme=[over]   length=4  Token is  WORD
Lexeme=[a]      length=1  Token is  WORD
Lexeme=[Lazy]   length=4  Token is  WORD
Lexeme=[Dog]    length=3  Token is  WORD
```

2.1.1 Lexical Specifications

In the previous section, we learnt that lexical analyser could be generated from a lexical specification file by using lexical analyser generator tools like *flex*. This section focuses on the composition of a lexical specification file.

A notation called the ***regular expressions*** is used to write lexical specifications. We shall first study the notation (Section 2.1.1.1), followed by an explanation of how a lexical specification file is structured (Section 2.1.1.2) and later write lexical specifications of C language (Section 2.1.2).

2.1.1.1 Regular Expressions A regular expression is a pattern that describes a set of strings. The simplest regular expression is the one that matches a single character. For example a regular expression '***s***' matches any input string where letter ***s*** is present like say *sink*, *ba**s**e*, ***s**tart*, *boo**s**t*, etc.

In order to understand the concepts of regular expressions (***RE's***), we shall use a utility '*egrep*' (***e**xtended **g**lobal **r**egular **e**xpression **p**rint*) available on UNIX, LINUX and other platforms. This utility can be used to try out various regular expressions and verify if the expected strings are matched or not. It is invoked in the following form:

```
$ egrep -n -e 'Regular Expression' file1 file2 ……
```

This utility searches file1, file2, file3, etc. for lines containing a match to the specified regular expression. We use a file '*my_input*' having the following lines as shown by the '*cat*' command for trying out various regular expressions throughout this section.

```
# -n option for showing line numbers
$ cat -n my_input
    1  sink
    2  base
    3  start
    4  dog
    5  goat
    6  boost
    7  easter
    8  ported
    9  global
   10  boss
   11  labs
```

We can try out the regular expression that matches a single character ***s*** for the file '*my_input*'

```
# Match any line with s
$ egrep -n -e 's' my_input
1:sink
2:base
3:start
6:boost
7:easter
10:boss
11:labs
```

Two regular expressions concatenated form a regular expression that recognises a match of first regular expression followed by a match of second. For example, a regular expression ***'st'*** matches any input where letter ***s*** is followed by a ***t*** like say *s**t**art*, *boo**st***, *ea**st**er*, etc.

```
# Character s followed by t
$ egrep -n -e 'st' my_input
3:start
6:boost
7:easter
```

There are some characters in the regular expressions, which have special meaning. They are called ***meta characters***. Table 2.1 gives a summary of all the meta characters used in regular expressions.

Table 2.1 *Meta characters in regular expressions*

Meta character	Description
.	Matches any character except a new line. For example, a regular expression ***'a.'*** matches 'a' followed by any character like b or c, etc. except a new line. Some strings that match the regular expression ***'a.'*** are 'b***at***', 'm***ad***', '***car***', etc.
^	Matches the start of the line. For example, a regular expression ***'^A'*** matches any line that starts with 'A'. Some lines that match regular expression ***'^A'*** are: ***A*** thing of beauty is a joy forever ***A*** fool and money are soon parted
$	Matches end-of-the line. For example, a regular expression ***'d$'*** matches any line that ends with 'd'. Some lines that match regular expression ***'d$'*** are: A friend in need is a friend indee***d*** "let's celebrate", the doctor sai***d***
[ ]	A character class—Matches any letter within the parenthesis. For example, a regular expression ***[012345]*** matches 0 or 1 or 2 or 3 or 4 or 5. Some strings that match this regular expression are 'var***1***', 'v***3***engine' and 'version***5***'. Note that 'v6' does not match the above regular expression. Within the parenthesis, the following characters have special meanings: A dash '–' inside a square bracket represents a range of characters to match, e.g. a regular expression ***[a–k]*** represents all the characters 'a', 'b', 'c' and so on till 'k'. Some strings that match the regular expression ***[a–k]*** are 'r***a***m', 'mo***b***', and 'mu***g***'. Note that 'mop' does not match the above regular expression.

	A circumflex '^' inside a square brackets represents the match of any character except the ones in the bracket, e.g. a regular expression ***[^abcd]*** represents all the characters except 'a' or 'b' or 'c' or 'd'. Some strings that match the above regular expression are '*pet*', '*fellow*'. The string *'bad'* does not match the regular expression.
|	Matches either the preceding regular expression or the succeeding regular expression. For example, a regular expression ***a|b*** represents match for 'a' or 'b'. Some strings that match the above regular expression are 'g***a***te', '***b***oost'. The strings 'fix' and 'group' do not match the regular expression.
( )	Used for grouping regular expressions. For example, the regular expression (ab) represents a match for 'a' followed by 'b'. The strings that match the above regular expression are 'c***ab***' and 'l***ab***'. The strings that do not match the above regular expression are 'garb', 'aerobics'.
* + ?	Unary operators for specifying repetition in regular expressions. * for zero or more, + for one or more and ? for zero or one. For example, the regular expression ***ab**** matches 'a' followed by no 'b' like '***a***ll', 'a' followed by one 'b' like '***ab***out' or a followed by any number of 'b's like 'g***abbbbbb***'. The regular expression ab+ matches 'a' followed by at least one 'b' like 'c***ab'*** or any number of 'b's like 'g***abbbbbb***' . Note that it does not match 'a' followed by no 'b' as in the previous case of regular expression ***'ab*'*** like say, ***a***ll. The regular expression ab? Matches 'a' followed by no 'b' like ***a***ll or at the most one 'b' like 'l***ab***'. It does not match the string 'jabb'.
{ }	Indicates how many times the previous pattern is matched. For example, the regular expression a{1,3} represents a match of one to three occurrences of 'a'. The strings that match the above regular expression are 'd***a***d', 'd***aa***d', and 'd***aaa***d'. The strings that do not match the above regular expression are 'sting' and 'out'.

Let's try out a few examples using the *egrep* utility and the file *'my_input'* shown earlier in this section to clarify the concepts of meta characters.

The regular expression ***'o.t'*** matches any input where letter ***o*** is followed by any character followed by ***t*** like say *g**oat**, b**oost**, p**ort**ed,* etc.

```
# o followed by any char then by t
$ egrep -n -e 'o.t' my_input
5:goat
6:boost
8:ported
```

The regular expression ***'^b'*** matches any input where letter ***b*** is the start of the line like say ***b**ase*, ***b**oost*, ***b**oss*, etc.

```
# b is the first letter in the line
$ egrep -n -e '^b' my_input
2:base
6:boost
10:boss
```

The regular expression ***'t$'*** matches any input where letter ***t*** is the end of the line like say *start,* *goat,* *boost.*

```
# t is the last letter in the line
$ egrep -n -e 't$' my_input
3:start
5:goat
6:boost
```

The regular expression ***'[ats]'*** matches any input where letters ***a*** or ***t*** or ***s*** are present like say *sink*, *base*, *start*, etc.

```
# Matches a or t or s
$ egrep -n -e '[ats]' my_input
1:sink
2:base
3:start
5:goat
6:boost
7:easter
8:ported
9:global
10:boss
11:labs
```

The regular expression ***'[p-s]'*** matches any input line where letter ***p*** or ***q*** or ***s*** exist like say *sink*, *base*, *ported*, etc.

```
# Matches p or q or s
$ egrep -n -e '[p-s]' my_input
1:sink
2:base
3:start
6:boost
7:easter
8:ported
10:boss
11:labs
```

The regular expression ***'[^b]a'*** matches any input where some letter other than 'b' is followed by 'a' like say *start*, *goat*, *easter*, *labs.*

```
# Matches any char other than b followed by a
$ egrep -n -e '[^b]a' my_input
3:start
5:goat
7:easter
11:labs
```

The regular expression ***'bo*s'*** matches any input line where ***b*** is followed by zero or more ***o*** and then a ***s*** like say ***boos***t, ***bos***s, la***bs.***

The regular expression ***'bo+s'*** matches any input line where ***b*** is followed by one or more ***o*** and then a ***s*** like say ***boos***t, ***bos***s.

The regular expression ***'bo?s'*** matches any input line where ***b*** is followed by zero or one ***o*** and then a ***s*** like say ***bos***s, la***bs.***

```
# Matches b followed by zero or any number of o then s
$ egrep -n -e 'bo*s' my_input
6:boost
10:boss
11:labs
# Matches b followed by one or any number of o then s
$ egrep -n -e 'bo+s' my_input
6:boost
10:boss
# Matches b followed by one or no o then s
$ egrep -n -e 'bo?s' my_input
10:boss
11:labs
# Matches b followed by 1 or 2 instances of o followed by s
$ egrep -n -e 'bo{1,2}s' my_input
6:boost
10:boss
```

The regular expression ***'oa|or'*** matches any input line where ***oa*** or ***ort*** exists like say *g**oa**t, p**ort**ed.*

```
# Matches oa or ort
$ egrep -n -e 'oa|ort' my_input
5:goat
8:ported
```

The regular expression 'The regular expression ***' (oa|os)t'*** matches any input line where ***oa*** or ***os*** followed by ***t*** exists like say *g**oat**, bo**ost.***

```
# Matches oa or os followed by t
$ egrep -n -e '(oa|os)t' my_input
5:goat
6:boost
```

Having understood the concepts of regular expressions, let's define the constructs of C language like identifiers, constants, etc. using regular expressions.

The C language keywords are the easiest ones to be described using regular expression. The regular expression is same as the keyword. For example, the keyword 'goto' is represented by a regular expression '*goto*', and 'switch' by regular expression '*switch*', and so on.

A C identifier begins with an alphabet or underscore, followed by either an alphabet or underscore or digit. A C identifier is represented by a regular expression `'[a-zA-Z_]([a-zA-Z_]|[0-9])*'`

An integer constant in C language has one or more digits followed optionally by a type qualifier like L or l or U or U. This can be represented by a regular expression `'[0–9]+(u|U|l|L)?'`

A hexadecimal constant in C language begins with 0 followed X or x followed by one or more number of hexadecimal digits. A type qualifier like u or U (for unsigned) or L or l (for Long) might optionally follow the hexadecimal digits. A hexadecimal constant can be represented by a regular expression `'0[xX][0-9a–fA F]+(u|U|l|L)?'`

The regular expression for both the C identifier and integer constant contain [0–9] in them. Similarly, `{u|U|l|L}` is contained in the regular expression for hexadecimal constant as well as integer constant. The repetition of the same regular expression in various places can be avoided by having a ***regular definition*** for that particular regular expression. Table 2.2 gives regular definitions for representing a digit, a integer specifier, letter, and so on.

Table 2.2 *Regular definitions*

Regular Definition	Regular expression for the definition
`DIGIT`	`[0–9]`
`IS`	`(u|U|l|L)`
`FS`	`(f|F|l|L)`
`LETTER`	`[a-zA-Z_]`
`EXPONENTIAL`	`[Ee][+-]?[0–9]+`

The regular definitions given in Table 2.2 can be used in describing the C identifier as `{LETTER}{LETTE R}|{DIGIT})*` which is more readable compared to the earlier version `'[a-zA-Z_]([a-zA-Z_]|[0-9])*'` even though both mean the same.

Table 2.3 gives the regular expression for a few C constructs, some of them using the regular definitions mentioned previously.

Table 2.3 *Regular expressions for a few C constructs*

Regular expression	Description
`"("`	Open parenthesis
`++`	Increment operator
`{DIGIT}+{IS}?`	Integer constant
`";"`	Semicolon
`"for"`	For keyword
`{LETTER}({LETTER}|{DIGIT})*`	C identifier
`0{DIGIT}+{IS}?`	Octal constant
`{DIGIT}+{EXPONENTIAL}{FS}?`	Exponential constant
`\"(\\.|[^\\"])*\"`	A string literal

Before we end this section, let us see some of the limitations of regular expressions.

Regular expressions are not suited to describe nested structures like C expressions. For example, consider a C expression `((my_var+1) == 10)` where *my_var* is a C identifier. This entire C expression cannot be described using regular expression, because it cannot determine the matching braces. The same can be described adequately using context-free grammar as we can see in the coming chapters. This is one of the main reasons that lexical analysis is separated from parsing.

Regular expressions can be used to denote only a fixed number of repetitions (using {x,y} notation) or an unspecified number of repetitions (using * operator) of given construct. It cannot be used in situations where the length of the regular expression needs to be deduced. For example, in FORTRAN, Hollerith format strings are used to print formatted output on the screen. It is shown below:

100 FORMAT (17H TITLE OF PROGRAM)

In the FORMAT statement above, 17 is the length of the string that follows, i.e. 'TITLE OF PROGRAM'. Regular expression cannot be used to recognise the FORMAT statement because the length of the regular expression cannot be deduced from the statement.

2.1.1.2 ***Structure of a Lexical Specification File*** In this section, we shall understand in detail about the format of lexical specification file. We will use the file *'words.l'* shown in Listing 2.1 as an example to explain the concepts clearly. Recall that this file was used earlier in Section 2.1 for creating a lexical analyser, which splits English sentences into words.

```
LETTER [A-Za-z]
WS [ \t\n]

%{
         #define WORD 1
%}

%%
{LETTER}+ { return (WORD) ; }

{WS}   { ; /* eat up White Space */ }
.      { ; /* eat up all others like punctuation marks etc. */}

%%

#include <stdio.h>
int main(int argc, char **argv)
{
      FILE *fp;
      int ret_val;

      if(argc != 2){
            printf(“Usage %s <input file>\n”,argv[0]);
            exit(1);
      }

      if((fp = fopen(argv[1],”r”)) == NULL ){
            printf(“File [%s] does not exist \n”,argv[1]);
            exit(1);
      }

32       yyin = fp; /* Input file to lexical analyser */
33
34       while (1) {
35             if( (ret_val=yylex()) == 0 ){
36                    break;
37             }
38
39
40             /* return value signifies token type */
41             if(ret_val == WORD ){
42                    printf("Lexeme=[%s] \t length=%d ",yytext,yyleng);
43                    printf(" \t Token is WORD \n");
44             }
45       }
46 }
```

Listing 2.1 *words.l*

A lexical specification (in short a lex) file consists of 3 parts:

```
Declarations
%%
Translation Rules
%%
Auxiliary Functions
```

The declarations section in *'words.l'* extends from lines 1 to 8. The declarations section consists of regular definitions that can be used in translation rules. This can be seen on lines 1 and 2 of the *'words.l'* where we specified regular definition for LETTER and WS. Apart from the regular definitions, the declaration section usually contains the #defines, C prototype declarations of the functions used in translation rules and some #include statements for the library functions used in translation rules. All the C statements mentioned above are enclosed in the special brackets %{and %}. This can be seen from lines 4 through 6 where the #define for WORD is done within special brackets. Anything appearing between these special brackets is copied verbatim into *lex.yy.c*

The translation rules section consists of statements in the following form:

```
Pattern1   { Action 1 }
Pattern2   { Action 2 }
Pattern3   { Action 3 }
```

where Pattern1, Pattern2,... Pattern *n* are all regular expressions, and the Action 1, Action 2,... Action *n* are all program segments describing the action to be taken when the pattern matches. The pattern is a regular expression that we learnt in the previous section. The action is typically a return statement indicating the type of token that has been matched as can be seen from line 9 of '*words.l*'. There are a few generated global variables that can be used in the action statements. For example, *yytext* contains the lexeme, *yyleng* gives the length of the lexeme. For the tokens that do not have any significance for the parser (like white space, new line, etc.) the action statement would not have a return statement as seen by lines 11 and 12.

The auxiliary functions section extends from line 15 till the end of the program. The auxiliary functions section usually contains the definition of the C functions used in the action statements. The whole section is copied "as is" into *lex.yy.c*. In *'words.l'* example, we have also defined the '*main( )'* in auxiliary section.

The code from line 34 to 45 is reflective of the parser to lexical analyser interaction. The function *yylex* is called repeatedly to get the next token of the input. The return value of *yylex* is indicates the type of token. A 0 return value signifies the end of input. We can observe that the *yylex* routine is called repeatedly to continue getting the next token until the end of the input.

2.1.2 A Lexical Analyser for C Language

In the last section we understood the lexical specification file format and how to generate a lexical analyser from lexical specification file using the tool *flex*. In Table 2.3, we saw how C constructs can be specified in regular expressions. In this section, we put these concepts into practice and generate a lexical analyser for C language program.

The following dialog shows the steps of converting the *'c-lex.l'* lexical specification file into a C lexical analyser. The C lexical analyser is shown splitting an input C program into tokens. The lexical analyser of the toy C compiler (mycc) that we are developing incrementally in this book is based on this lexical specification.

```
# Compiling the Lexical Specs to generate lex.yy.c
$ flex c-lex.l

# Compiling the lex.yy.c to generate a binary
$ gcc lex.yy.c -o lex_analyzer -lfl

# Input C program for tokenizing
$ cat -n test2.c
    1 int
    2 main ()
    3 {
    4    int i;
    5
    6    /* This is a comment.This will be stripped by Lexical Analyzer */
    7
    8    for (i = 0; i < 10; i++) {
    9        printf ("Hello World\n");
   10    }
   11 }

# Lexical Analyzer Tokenizing the input C Program
$ ./lex_analyzer test2.c
Lexeme=[int]        Length=3    Token is INT
Lexeme=[main]       Length=4    Token is IDENTIFIER
Lexeme=[(]          Length=1    Token is LP
Lexeme=[)]          Length=1    Token is RP
Lexeme=[{]          Length=1    Token is LC
Lexeme=[int]        Length=3    Token is INT
Lexeme=[i]          Length=1    Token is IDENTIFIER
Lexeme=[;]          Length=1    Token is SEMI
Lexeme=[for]        Length=3    Token is FOR
Lexeme=[(]          Length=1    Token is LP
Lexeme=[i]          Length=1    Token is IDENTIFIER
Lexeme=[=]          Length=1    Token is EQUAL
Lexeme=[0]          Length=1    Token is CONSTANT
Lexeme=[;]          Length=1    Token is SEMI
Lexeme=[i]          Length=1    Token is IDENTIFIER
Lexeme=[<]          Length=1    Token is LT
Lexeme=[10]         Length=2    Token is CONSTANT
Lexeme=[;]          Length=1    Token is SEMI
Lexeme=[i]          Length=1    Token is IDENTIFIER
Lexeme=[++]         Length=2    Token is INC_OP
Lexeme=[)]          Length=1    Token is RP
Lexeme=[{]          Length=1    Token is LC
Lexeme=[printf]     Length=6    Token is IDENTIFIER
Lexeme=[(]          Length=1    Token is LP
Lexeme=["Hello World\n"] Length=15 Token is STRING_LITERAL
Lexeme=[)]          Length=1    Token is RP
Lexeme=[;]          Length=1    Token is SEMI
Lexeme=[}]          Length=1    Token is RC
Lexeme=[}]          Length=1    Token is RC
```

2.2 THE MECHANICS OF LEXICAL ANALYSER GENERATORS

In the previous section, we understood how a lexical analyser generator like *flex* could transform lexical specifications of a language into a lexical analyser. The rest of the chapter discusses the concepts and algorithms that lexical analyser generators like flex would use to generate the lexical analyser from the lexical specifications.

The lexical analyser generated by a lexical analyser generator tools like *flex* can be broadly divided into 2 components

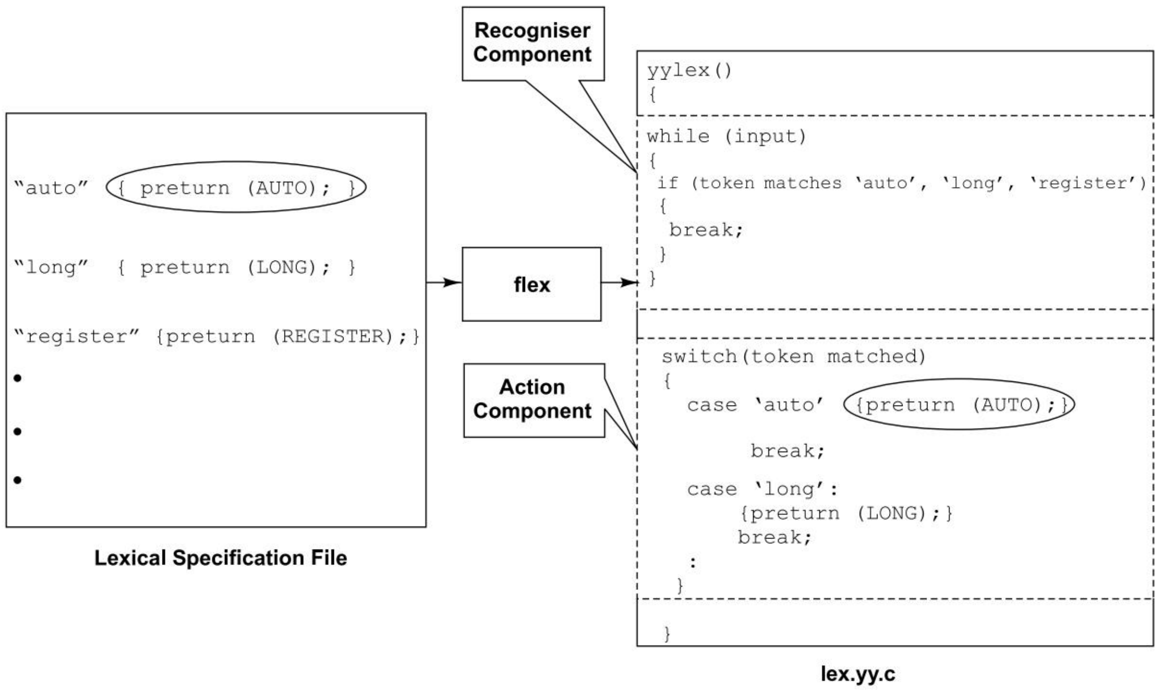
(1) A ***recogniser*** component, to recognise all the input strings that match the regular expressions specified in the translation rules of the lexical specifications file.

(2) An ***action*** component, which is the manifestation of the action, specified in the translation rules on recognising a specified pattern.

Figure 2.5 shows the idea of a recogniser component and action component in a lexical analyser generated by flex. The generated code is shown in a pseudo-code manner.

The recogniser component is built by converting the regular expressions given in the lexical specification file into a ***finite state machine (FSM)***. The next section (Section 2.2.1) discusses in detail what an FSM is and how it can be used to determine if the input matches any of the regular expression. The crux of the lexical analyser generator lies in conversion of the RE into FSM and using it to check if the input matches the RE.

The action component is a straightforward copy of the actions mentioned in the lexical specifications file.



```
"auto"     { preturn (AUTO); }

"long"     { preturn (LONG); }

"register" {preturn (REGISTER);}
•
•
•
```

```
yylex()
{
while (input)
{
 if (token matches 'auto', 'long', 'register')
 {
  break;
 }
}

 switch(token matched)
 {
   case 'auto' {preturn (AUTO);}
        break;
   case 'long':
       {preturn (LONG);}
       break;
    :
  }
}
```

Fig. 2.5 *Mechanics of lexical analyser generator*

2.2.1 Finite State Machines—DFA and NFA

In the context of lexical analysis, a finite state machine is a mechanism used to recognise a particular pattern in a given input. For example, a finite state machine can be used to recognise keywords like '*while*' and '*for*' in a given C program.

A finite state machine consists of:

- A finite number of states.
- A set of transitions from one state to another on the receipt of inputs, e.g. transition from state 1 to state 2 on receiving an input of 'A', state 1 to state 3 on input of 'B', etc.
- A start state.
- A set of accepting states, each signifying the successful recognition of input as a token.

A finite state machine can be visually represented by a labelled directed graph called a ***transition diagram***. The transition diagram shown in Fig. 2.6 represents the finite state machine, which recognises the C language keywords *for* and *while*.

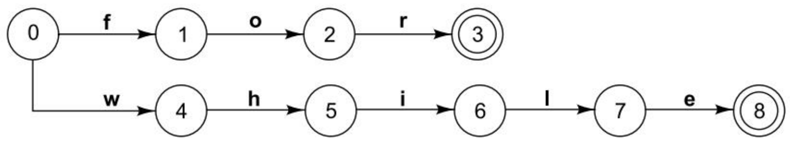


Fig. 2.6 *Transition diagram for keywords 'for' and 'while'*

Each state is depicted using a circle. Receiving a certain input causes a transition from one state to another. For example, while in state 1, on receiving an input ‘o’, there would be a transition from state 1 to state 2. The edge represents the character in the input based on which the transition between states happen. For example, the transition between state 4 and state 5 can only happen if the input is ‘h’. The states with concentric circles are called as ***accepting states***. They signify a successful recognition of a particular input string as a token. For example, reaching state 8 signifies that the input contained a *‘while’* in it.

A finite state machine can also be represented by a ***transition table***, in which each row represents a state and each column an input. The transition table equivalent of Fig. 2.6 is shown in Table 2.4. By observing the row corresponding to state 0, we can conclude that in state 0 an input of ‘w’ would cause a transition to state 4, while an input of ‘f’ in state 0 would cause a transition to state 1. Similarly, observing the row corresponding to state 1, we can conclude that an input of ‘o’ would cause a transition to state 2. In state 1, an input of ‘e’ is not defined, hence represented by a null (–) transition.

Table 2.4 *Transition table*

State	Input Symbol							
	e	**f**	**h**	**i**	**l**	**o**	**r**	**w**
0	–	1	–	–	–	–	–	4
1	–	–	–	–	–	2	–	–
2	–	–	–	–	–	–	3	–
3	–	–	–	–	–	–	–	–
4	–	–	5	–	–	–	–	–
5	–	–	–	6	–	–	–	–
6	–	–	–	–	7	–	–	–
7	8	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–

A transition diagram of the type shown in Fig. 2.6 can be implemented by using 3 data structures:

- A variable holding the current state.
- Transition table, a two-dimensional array for computing the next state. The next state is computed by indexing the table on the basis of current state and the input character. In other words, *next_state* = *transition_table[current_state][input_char]* using the C language syntax of a two-dimensional array.
- Accept marker, a single-dimensional array indexed by current state used to determine if the state is an accepting state or not. The accept marker array for the transition table shown in Table 2.4 is shown below. Observe that accept[3] and accept[8] are 1, signifying that they are accepting states.

–	–	–	1	–	–	–	–	1

The following dialog shows an implementation of a finite machine that recognises the keywords ‘for’ and ‘while’ using the data structures just discussed.

```
# Compiling trans.c to get a Binary
$ gcc -Wall trans.c -o trans

# Sample input file
```

```
$ cat -n test3.c
     1  int
     2  main ()
     3  {
     4     int i = 0;
     5
     6     while (i < 5) {
     7        printf ("Hi\n");
     8        i++;
     9     }
    10
    11     for (i = 0; i < 10; i++) {
    12        printf ("Hello World\n");
    13     }
    14  }(

# Tokenizing the input C program
$ ./trans test3.c
found 'while' Return Val=8
found 'for' Return Val=3
```

The finite state machine that we saw till now is called as ***deterministic finite automaton*** or ***DFA***. It is called deterministic because the next state can be determined by knowing the current state and the next input character.

There is another type of finite state machine called as ***non-deterministic finite state machine*** or ***NFA***. An NFA differs from the DFA in the following aspects:

An empty transition denoted by special symbol ∈ — *epsilon* is possible in NFA. This transition (also called as epsilon transition) can be made on empty string without advancing input. For example, the transition diagram shown in Fig. 2.6 is modified in Fig. 2.7 to show the epsilon transition. The transitions 0 to 1 and 0 to 5 do not require any input.

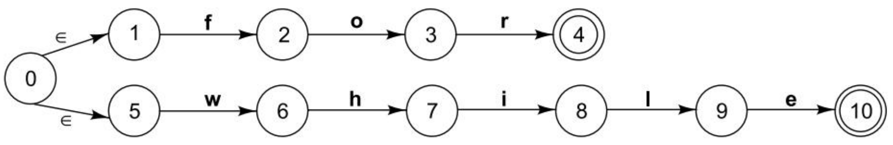


Fig. 2.7 *NFA transition diagram showing epsilon transition*

An NFA has no limitations on the number and type of edges. Two outgoing edges can have the same label. For example, the keywords *char* and *case* of C language can be recognised by a NFA shown in Fig. 2.8. Note that transitions 0 to 1 and 0 to 5 can happen on input 'c'. Another NFA that can recognise the C language keywords *case* and *char* is shown in Fig. 2.9. As you would understand in the next few sections, it is easier to convert an RE to the kind of NFA shown in Fig. 2.9 rather than Fig. 2.8.

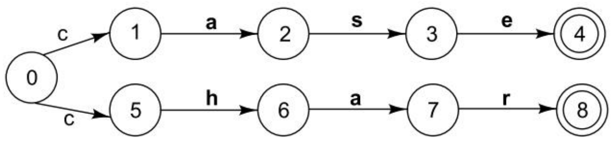


Fig. 2.8 *NFA can have two outgoing edges having same label 'c'*

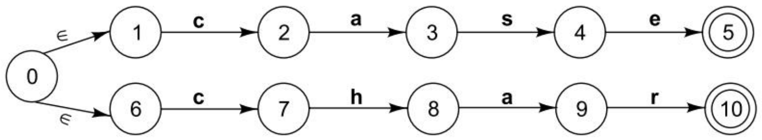


Fig. 2.9 *NFA recognising C keywords* **'***char***'** *and* **'***case***'**

In DFA, the next state can be determined by having the current state and the input character. In NFA, the *next set of states* can be determined by knowing the *current set of states* and the next input character. For example, the NFA shown in Fig. 2.9, if the current state is {0}, then on receiving an input of 'c', the next set of states is {2,7} (recall that ∈ transition does not need any input). This can be represented mathematically by:

next({0},'c') = {2,7}

Subsequent to that if another input say 'h' is received then the next set of states is {8}. This can be represented mathematically by:

next({2,7},'h') = {8}

Some other advanced aspects on which NFA and DFA can be compared are listed in Table 2.5.

Table 2.5 *NFA v/s DFA comparison*

NFA	DFA
A regular expression can be easily converted to NFA by using Thompson's construction.	A regular expression can be converted to DFA, but it is complex. The usual method for deriving a DFA from a regular expression is to first convert the RE to NFA and then translate the NFA to DFA.
NFA is optimised on space. However, it requires more computation to be done in order to verify if the input matches an RE.	The DFA is optimised in time, but it requires more memory for storing the state information. This is due to the number of DFA states being greater than the number of states of a corresponding NFA.
The time taken to recognise a string matching a regular expression using the RE's NFA increases, when the length of the RE increases. For example, the time taken to recognise a string matching a regular expression '*ab*|*cd*|*ef*' is greater than '*ab*|*cd*', which is greater than '*ab*'.	The time taken to recognise a RE using its DFA is independent of the length of RE. For e.g. the time taken to recognise a string matching a regular expression '*ab*|*cd*|*ef*' is same as '*ab*|*cd*', which is same as '*ab*' This is a very important characteristic that weighs heavily in favour of using DFA in lexical analysers.

2.2.2 From Lexical Specifications to Lexical Analyser

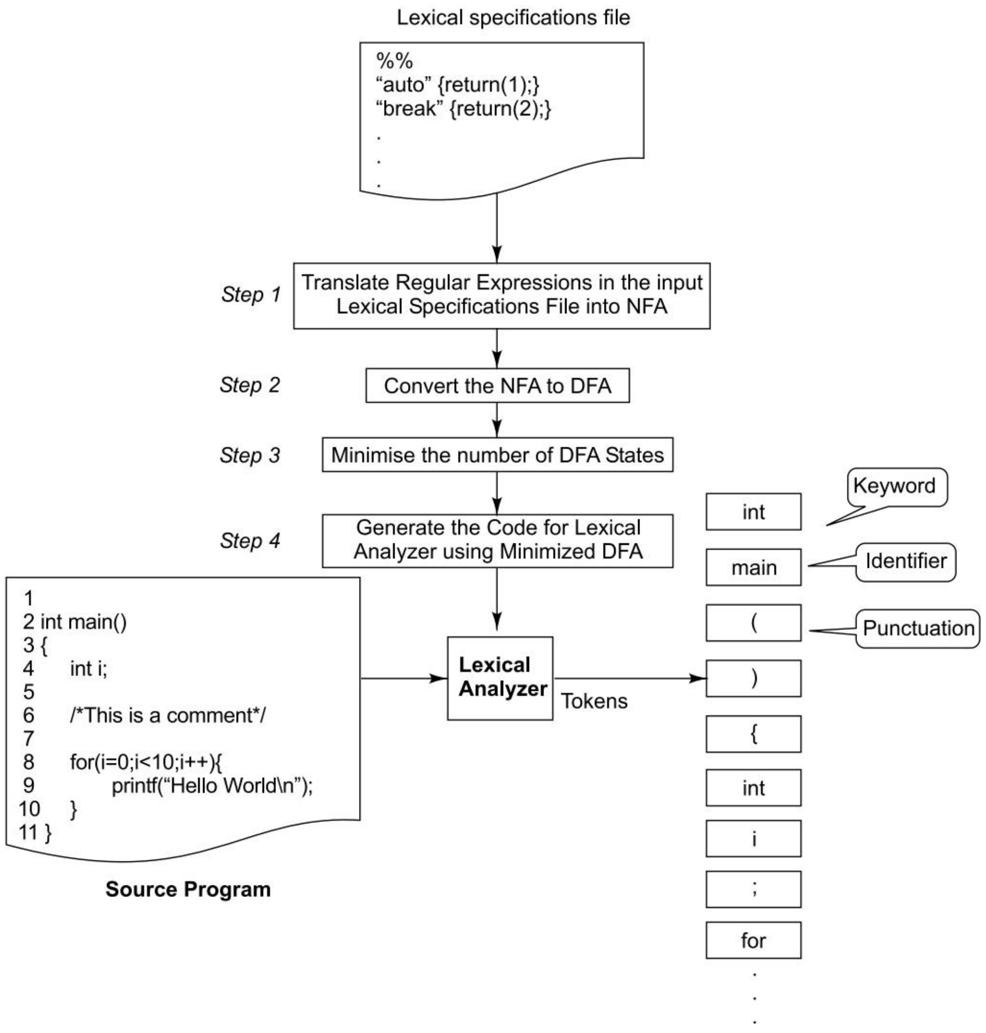
The recogniser component of the lexical analyser (Fig. 2.5) is built by converting the regular expressions given in the lexical specification file into a finite state machine. The finite state machine could be an NFA or a DFA. The choice is DFA when speed is important and there are no memory constraints. In other cases where memory is at premium and speed can be compromised, the choice is NFA. In practice, most of the lexical analyser generators including *flex* base the recogniser component on a DFA.

Typically, lexical analyser generators convert the input lexical specification file into a lexical analyser in the following 4 steps as shown in Fig. 2.10.

Step 1: Translate the regular expressions in the input lexical specifications file into NFA.

Step 2: Convert the NFA into DFA.

Step 3: Minimise the number of DFA states.
Step 4: Generate the code for the lexical analyser using the minimised DFA.



Lexical specifications file
%%
"auto" {return(1);}
"break" {return(2);}
Step 1
Translate Regular Expressions in the input Lexical Specifications File into NFA
Step 2
Convert the NFA to DFA
Step 3
Minimise the number of DFA States
Step 4
Generate the Code for Lexical Analyzer using Minimized DFA
Lexical Analyzer
Tokens
Source Program
Keyword
Identifier
Punctuation
int
main
(
)
{
int
i
;
for

```
1
2 int main()
3 {
4     int i;
5
6     /*This is a comment*/
7
8     for(i=0;i<10;i++){
9           printf("Hello World\n");
10    }
11 }
```

Fig. 2.10 *Stages of lexical analyser generator*

In step 1 of Fig. 2.10, the regular expressions given in the lexical specification file are converted into NFA by using an algorithm called as Thompson's construction. The second step converts the NFA to DFA. The memory required to implement a DFA depends on the number of states in the DFA. The DFA obtained in step 2 is usually not optimised in terms of the number of states and hence might turn out having a higher memory requirement. In order to reduce the memory needed, the number of states in DFA is minimised using several techniques in Step 3. Finally using the minimised DFA, the code is generated for the lexical analyser in step 4.

2.3 RLEX—A RESTRICTED LEXICAL ANALYSER GENERATOR

This section is devoted to understanding the 4 steps shown in Fig. 2.10 for converting lexical specifications into a lexical analyser. In the process of grasping the 4 steps, we build a toy lexical analyser generator called ***rlex (restricted lexical analyser).*** rlex is a lexical analyser generator similar to 'flex', but with a restricted feature set in the interest of keeping it simple and focused on the concepts of lexical analysis. The development of *rlex* is done in step-by-step fashion to explain concepts and show relevant examples.

Each one of the next four sub-sections represents a step in making of a lexical analysis generator. While Sub-section 2.3.1 represents the step1 of lexical analyser generator, i.e. translation of regular expression to NFA, Sub-section 2.3.2 discusses the second step of lexical analyser generator, which is the conversion of the NFA to DFA, and so on.

2.3.1 Translating Regular Expression to NFA

The algorithm to convert a regular expression to NFA was first given by Ken Thompson, when he was working on the QED editor at Bell Labs.

The first step in the algorithm is to break up the input regular expression into smaller components that are easier to process independently.

Consider the regular expression ***abc*ef|ij(kl)?*** for discussion, this is broken up into:

1. ***Terminal symbols*** that individually represent an atomic unit of regular expression. For example, in the above regular expression *a, b, c, e,* etc. represent terminal symbols.
2. ***Factor***, which represent the grouping of a terminal symbol with a closure operator like **, ?,* + etc. For example, in the above regular expression *c** represents a factor.
3. ***Sub-expressions***, which themselves constitute a single regular expression. For example, in the above RE, *ef* is a sub-expression, *ij* is another sub-expression, *(kl)* is another sub-expression.

The process of splitting the input into smaller components as explained above, is illustrated in Fig. 2.11.

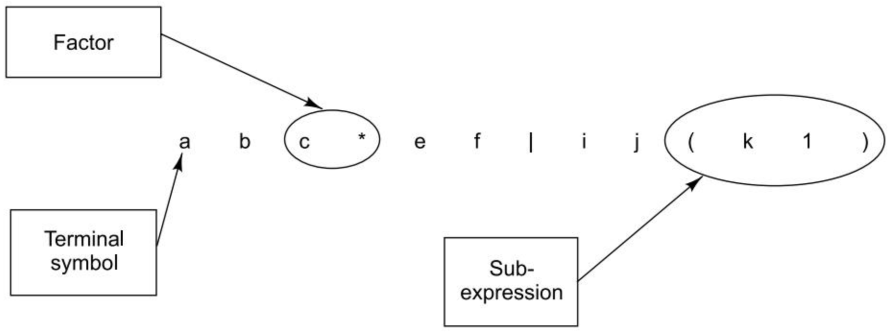
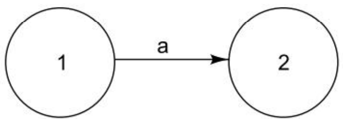


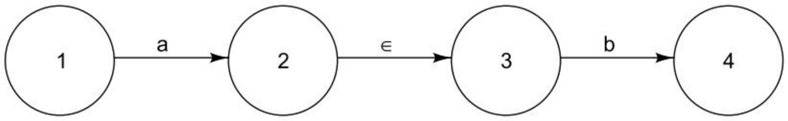
Fig. 2.11 *Parsing a regular expression*

After the regular expression is broken up into smaller components, viz. terminal symbols, factor and sub-expressions, NFA's are created for each of those using the following rules.

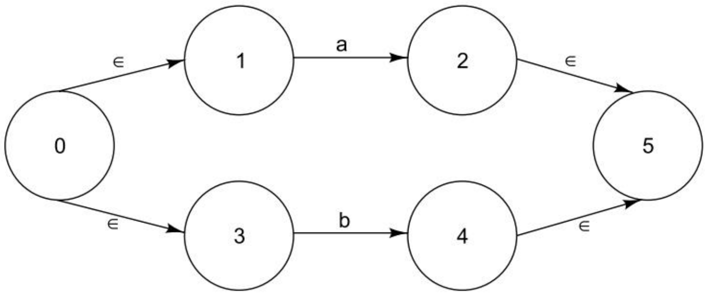
(1) For a terminal symbol, the NFA is shown below:



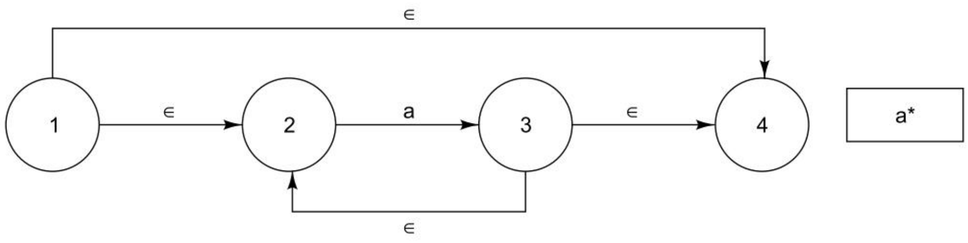
(2) For concatenation of two regular expressions, for example, *ab*, the NFA is shown below:

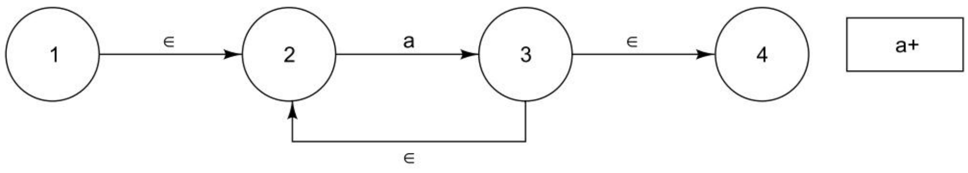


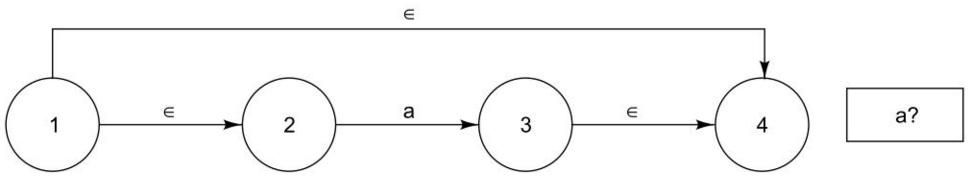
(3) For alternation of two regular expressions, for example, *a*|*b*, the NFA is shown below:



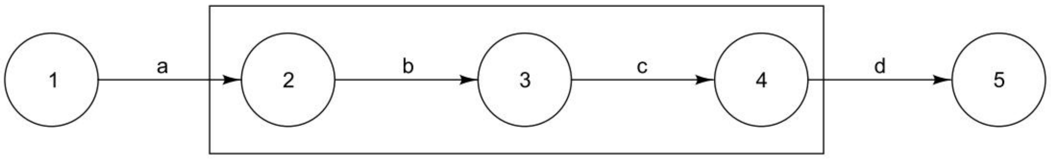
(4) For a factor, the NFA is shown below:







(5) For a sub-expression, the NFA is similar to concatenation shown earlier, excepting the fact that the whole of sub-expression needs to be considered as a single unit.



The translation of a regular expression to NFA involves 2 steps.

1. Parse the regular expression to break it up to terminal symbols, factor and sub-expressions.
2. Use the rules given above to formulate NFA for the regular expression.

In step (a), we break up the input RE after parsing it character by character into terminal symbols, factor or sub-expressions. The step (b) uses the rules given above and makes the NFA. The following example is an implementation of the above 2 steps for converting a regular expression into an NFA.

2.3.1.1 *Example 1—Translation of Regular Expression to NFA*

This section demonstrates an example program that can convert a given regular expression into NFA using the theory described in the preceding section. The program takes a regular expression on the command line, converts it into an NFA and prints out the details of the NFA. The output of the program is in the form of translation table that we discussed earlier. The program is also capable of taking a single lexical specification translation rule (recall Section 2.1.1.2) and printing out the details of the resultant NFA. The dialog below shows the example program taking in different regular expressions, and printing out the NFA details.

```
# Making the example
$ g++ -g -Wall rlex.cc ex1.cc -o ex1

# The argument is any regular expression
$ ./ex1 'abcd'

               NFA Transition Table

============|===============================
State       | Input symbol
            | a  b    c    d    epsilon
============|===============================
START       0  | -    -    -    -    { 1 }
            1  | 2    -    -    -    -
            2  | -    -    -    -    { 3 }
            3  | -    4    -    -    -
            4  | -    -    -    -    { 5 }
            5  | -    -    6    -    -
            6  | -    -    -    -    { 7 }
            7  | -    -    -    8    -
ACCEP       8  | -    -    -    -    -
 ==========  |=============================

# Another regular expression
$ ./ex1 '(a|b|c)?d'

               NFA Transition Table
```

```
============|================================
State          | Input symbol
               |a    b    c    d    epsilon
===========|================================
START       0  |-    -    -    -    { 11 }
            1  |2    -    -    -    -
            2  |-    -    -    -    { 6 }
            3  |-    4    -    -    -
            4  |-    -    -    -    { 6 }
            5  |-    -    -    -    { 1 3 }
            6  |-    -    -    -    { 10 }
            7  |-    -    8    -    -
            8  |-    -    -    -    { 10 }
            9  |-    -    -    -    { 5 7 }
           10  |-    -    -    -    { 12 }
           11  |-    -    -    -    { 9 12 }
           12  |-    -    -    -    { 13 }
           13  |-    -    -    14   -
ACCEP      14  |-    -    -    -    -
============|================================

# Another regular expression
$ ./ex1 '(a|b)*abb'

       NFA Transition Table

===========|======================
State         | Input symbol
              | a   b   epsilon
===========|======================
START      0  | -   -   { 7 }
           1  | 2   -   -
           2  | -   -   { 6 }
           3  | -   4   -
           4  | -   -   { 6 }
           5  | -   -   { 1 3 }
           6  | -   -   { 8 5 }
           7  | -   -   { 5 8 }
           8  | -   -   { 9 }
           9  | 10  -   -
          10  | -   -   { 11 }
          11  | -   12  -
          12  | -   -   { 13 }
          13  | -   14  -
   ACCEP  14  | -   -   -
===========|======================

# The argument can be a Translation rule of a Lex file
$ ./ex1 'ab printf("ab found");'

       NFA Transition Table
```

```
==========|=====================
State        | Input symbol
             | a b epsilon
==========|=====================
START     0  | - -     { 1 }
          1  | 2 -     -
          2  | - -     { 3 }
          3  | - 4     -
ACCEP  4 | - -    - Accept Action is [ {printf("ab found");}]
==========|=====================
```

2.3.1.2 ***Interpreting an NFA*** In Section 2.3.1.1, we looked at how to convert a regular expression into an NFA, which is the first step in Fig. 2.10 lexical analyser generator. In this section, we examine how to use the NFA created from regular expression to recognise if a given input string matches the regular expression or not.

Consider a regular expression ***'(a|b)*abb'*** for discussion. The NFA for it can be derived using the earlier example—*ex1* as shown in the dialog below:

```
# The argument is any regular expression
$ ./ex1 '(a|b)*abb'

            NFA Transition Table

============|=====================
State      | Input symbol
              |   a    b    epsilon
===========   |=====================
START    0 |   -   - { 7 }
            1 |   2    -    -
            2 |   -    -    { 6 }
            3 |   -    4    -
            4 |   -    -    { 6 }
            5 |   -    -    { 1 3 }
            6 |   -    -    { 8 5 }
            7 |   -    -    { 5 8 }
            8 |   -    -    { 9 }
            9 |  10    -    -
           10 |   -    -    { 11 }
           11 |   -   12    -
           12 |   -    -    { 13 }
           13 |   -   14    -
 ACCEP  14 |   -   -    -
============|=====================
```

Figure 2.12 shows the graphic view of NFA constructed from the output of *ex1*.

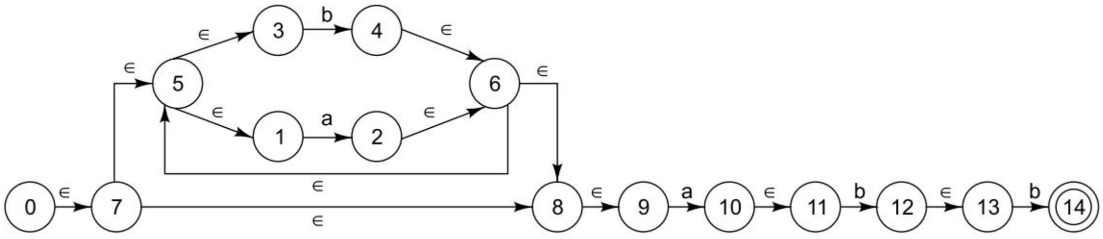


Fig. 2.12 *NFA for '(a|b)*abb'*

In an NFA, at any given point there are '*n*' states waiting in parallel for input. This is due to the fact that an ∈ transition can be taken without input. For example, in the NFA shown in Fig. 2.12, the start state is 0. The ∈ transitions that can be taken from state 0 without having any input are:

$0 \rightarrow \mathbf{7}$

$0 \rightarrow 7 \rightarrow \mathbf{5}$

$0 \rightarrow 7 \rightarrow 5 \rightarrow \mathbf{3}$

$0 \rightarrow 7 \rightarrow 5 \rightarrow \mathbf{1}$

$0 \rightarrow 7 \rightarrow \mathbf{8}$

$0 \rightarrow 7 \rightarrow 8 \rightarrow \mathbf{9}$

$0 \rightarrow \mathbf{0}$

Thus, the NFA States that can be reached from 0 on ∈ transitions alone are {7,5,3,1,8,9,0}. Note that we include the state from which the ∈ transitions originate (state 0 in this case) automatically. Figure 2.13 shows the NFA states (shaded) that are reachable on ∈ transitions alone. This set of states that are waiting for input in parallel, due to the epsilon transitions alone is called *as* ***∈-closure set***. It is denoted by e-closure(s), where s is a state. Thus,

```
∈-closure ({0}) = { 0 7 5 3 1 8 9 }
```

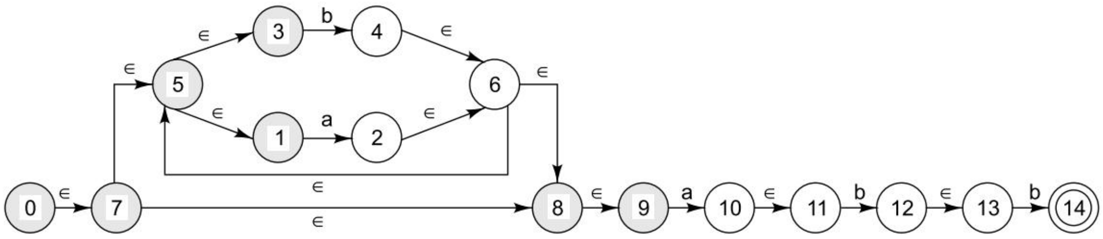


Fig. 2.13 *Illustration of ∈closure set*

On receiving an input symbol in a particular state, we need to apply the transition to all of the NFA states waiting in parallel at that point. For example, consider an input symbol 'a' received in the start state for the above NFA, the NFA states waiting in parallel as seen above are `{0 7 5 3 1 8 9}`. We should apply the transition to all of these states. Out of these states, only state 1 and state 9 have a valid transition for the input character 'a'. Applying the transition for 'a' on these states, we get state 2 and 10 respectively as shown in Fig. 2.14. This set of NFA States `{2,10}`, which has been derived by applying a transition for an input symbol to all of the NFA states waiting in parallel, is called as ***move set***. Formally stated, *move (T, a)* represents the set of NFA states to which there exists a transition on input symbol 'a' from some NFA state s in T.

```
move_set({ 0 7 5 3 1 8 9 }, a ) = {2,10}
```

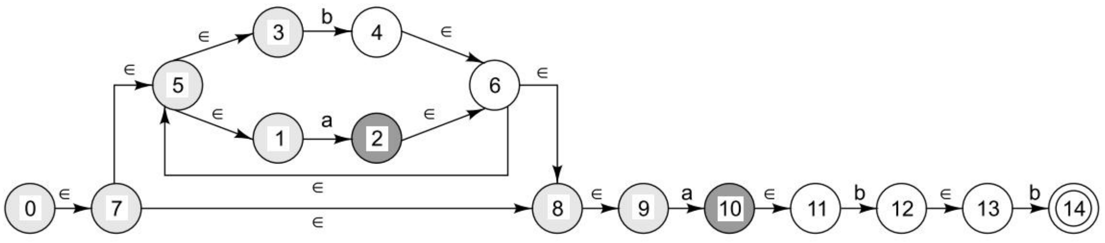


Fig. 2.14 *Illustration of move set*

2.3.1.3 ***Algorithm for Interpreting an NFA*** We shall use the concepts of closure set and move set to devise the algorithm for interpreting an NFA.

The algorithm for interpreting an NFA is presented in Algorithm 2.1. The basic strategy of the algorithm is to:

1. Identify all the NFA States waiting in parallel. This is done by computing the e closure set.
2. Take in the next input character and apply the transition on all of the NFA States waiting in parallel. This is done by computing the move set and then applying epsilon closure on it.
3. If the resulting NFA state out of applying transition in step 2, is an accepting state return a SUCCESS. If the resultant NFA State is not an accepting state, then go to step 2.

```
start_state is the start state of the NFA
cur_ptr is a pointer to the line of text being checked for match with the regular expression
nfa_accept_set is the set containing all the accept states in the NFA
```

```
interpret_nfa()
{
          current_set = ∈-closure(start_state)
          start_set = current_set ;
          while((c= *cur_ptr) != End-of-line){
                    move_set=move(current_set,c)
                    e_closure_set=e_closure(move_set)
                    if(e_closure_set is not empty set){
                           If(e_closure_set ∩ nfa_accept_set){
                                  Return(SUCCESS) ;
                           }
                           current_set = e_closure_set;
                    } else {
                           if(current_set != start_set){
                                  current_set = start_set;
                                  continue;
                           }
                    }
                    cur_ptr ++ ;
          }
          return (FAILURE) ;
}
```

Algorithm 2.1 *Interpreting NFA*

Let us see the algorithm at work given an input string '***aabb***' for the NFA shown in Fig. 2.12. Initially the NFA states that are waiting in parallel for input is given by:

```
e_closure( {0} ) = { 0 1 3 5 7 8 9 }                    ............................. (I)
```

On consuming the first 'a' of the input, the NFA states that are waiting for input in parallel is determined by

```
move({ 0 1 3 5 7 8 9 },a)= { 2 10 }
e_closure({ 2 10 })= { 1 2 3 5 6 8 9 10 11 }            ............................. (II)
```

On consuming the second 'a' of the input, the NFA states that are waiting for input is determined by

```
move({ 1 2 3 5 6 8 9 10 11 },a)= { 2 10 }
e_closure({ 2 10 })= { 1 2 3 5 6 8 9 10 11 }            ............................. (III)
```

On consuming the next character of the input 'b', the NFA states that are waiting for input is determined by

```
move({ 1 2 3 5 6 8 9 10 11 },b)= { 4 12 }
e_closure({ 4 12 })= { 1 3 4 5 6 8 9 12 13 }            ............................. (IV)
```

On consuming the next character of the input 'b', the NFA states that are waiting for input is determined by

```
move({ 1 3 4 5 6 8 9 12 13 },b)= { 4 14 }
e_closure({ 4 14 })= { 1 3 4 5 6 8 9 14 }               ............................. (V)
```

The accept state of 14 is found in (V), thus signalling the match of the input to the RE.

Programs such as these that can read a string as an input and output SUCCESS, if the string is a sentence in the language or failure, if it is not are called as ***recognisers***. The egrep utility that we used in Section 2.1.1.1 is a good example of a recogniser.

2.3.1.4 *Example 2—Interpreting NFA*

This section demonstrates an example program that is similar to the functionality of *egrep* utility. The program implements Algorithm 2.1 for interpreting the NFA and reporting if the input string matches the regular expression or not. The program takes in a regular expression and file name to search for a string that matches the regular expression. The output of the program is the lines in the file matching the regular expression along with the line numbers. The dialog below shows the example program taking in different regular expressions and file names, and printing out the lines in the file matching the regular expression.

```
# Making the example
$ g++ -g -Wall rlex.cc ex2.cc -o ex2

# Use the file that we used before for learning about the regular expressions
$ cat -n my_input
    1  sink
    2  base
    3  start
    4  dog
    5  goat
    6  boost
    7  easter
    8  ported
    9  global
   10  boss
   11  labs

# Matches b followed by zero or any number of o then s
```

```
$ ./ex2 'bo*s' my_input
6:boost
10:boss
11:labs

# Matches b followed by one or any number of o then s
$ ./ex2 'bo+s' my_input
6:boost
10:boss

# Matches b followed by one or no o then s
$ ./ex2 'bo?s' my_input
10:boss
11:labs

# Matches oa or ort
$ ./ex2 'oa|ort' my_input
5:goat
8:ported

# Matches oa or os followed by t
$ ./ex2 '(oa|os)t' my_input
5:goat
6:boost

# Verbose mode
# Test file containing a string
$ cat -n test4
       1 aabbb

$ ./ex2 -v '(a|b)*abb' test4

       NFA Transition Table

===========|=====================
State      | Input symbol
           |  a    b    epsilon
===========|=====================
START   0  | -    -      { 7 }
        1  | 2    -      -
        2  | -    -      { 6 }
        3  | -    4      -
        4  | -    -      { 6 }
        5  | -    -      { 1 3 }
        6  | -    -      { 8 5 }
        7  | -    -      { 5 8 }
        8  | -    -      { 9 }
        9  | 10   -      -
       10  | -    -      { 11 }
       11  | -    12     -
       12  | - - { 13 }
       13   | - 14     -
ACCEP  14   | - - -
===========|=====================
e_closure({ 0 }) = { 0 1 3 5 7 8 9 }
move({ 0 1 3 5 7 8 9 },a)= { 2 10 }
e_closure({ 2 10 })= { 1 2 3 5 6 8 9 10 11 }
move({ 1 2 3 5 6 8  9 10 11 },a)= { 2 10 }
e_closure({ 2 10 })= { 1 2 3 5 6 8 9 10 11 }
move({ 1 2 3 5 6 8 9 10 11 },b)= { 4 12 }
e_closure({ 4 12 })= { 1 3 4 5 6 8 9 12 13 }
move({ 1 3 4 5 6 8 9 12 13 },b)= { 4 14 }
e_closure({ 4 14 })= { 1 3 4 5 6 8 9 14 }
Accept State =14
1:aabbb
```

2.3.2 Converting NFA to DFA

In Section 2.3.1, we learnt how to convert a regular expression to NFA and interpret an NFA to recognise strings that match a regular expression. While the NFA can be used to recognise strings matching the regular expression, there is an inherent disadvantage. The time taken to recognise an input string using the NFA is dependent on the number of NFA states. The time taken to determine if an input string x, matches a regular expression 'r' is proportional to length of 'x' multiplied by length of 'r'. Due to this, complex regular expressions take longer time to recognise input string if the NFA is used. This is the reason why NFA is not used commonly for recognising the strings. The DFA does not have this problem. Hence NFA is converted to DFA and used for recognising input strings matching regular expression. The technique of converting the f is explained in this section.

This represents the second step (conversion of NFA to DFA) in the development of a lexical analyser generator shown in Fig. 2.10.

Consider the regular expression ***'(a|b)*abb'*** whose NFA was discussed in section 2.3.1.2. We shall try and convert the same NFA into DFA. The NFA is reproduced here for discussion.

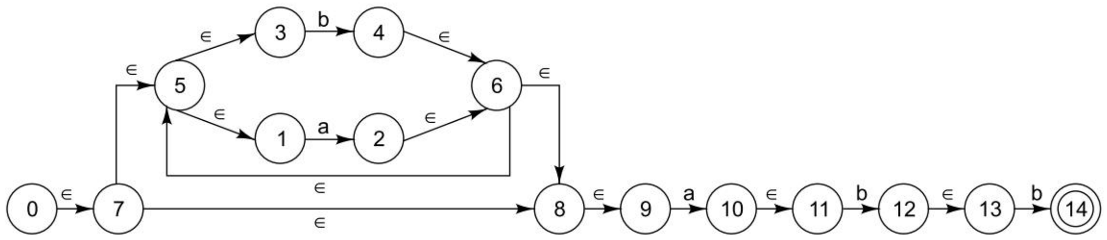


Fig. 2.15 *NFA for (a|b)*abb*

The basic principle on which the NFA to DFA conversion works is that each DFA state corresponds to a set of NFA states that are waiting in parallel for input.

Initially the NFA states that are waiting in parallel for input is given by

```
    e_closure( {0} ) = { 0 1 3 5 7 8 9 }        ( DFA State 0 )
```

This NFA set `{ 0 1 3 5 7 8 9 }` is the starting DFA state (state 0).

On scanning the regular expression '*(a|b)*abb*' , it can be concluded that the set of all possible input characters that can result in transitions leading to recognition of the input are {a b}. The next few DFA states are derived by simulating the inputs {a b}. Thus,

Applying transitions of a and b on DFA state 0

```
move ( { 0 1 3 5 7 8 9 } ,a )= { 2 10 }
e_closure ( { 2 10 })= { 1 2 3 5 6 8 9 10 11 }       ( DFA State 1 )
move ( { 0 1 3 5 7 8 9 } ,b )= { 4 }
e_closure ( { 4 })= { 1 3 4 5 6 8 9 }                 ( DFA State 2 )
```

The transition table for these 2 transitions can be represented by:

```
dtran[0,'a'] = 1
dtran[0,'b'] = 2
```

Applying transitions of a and b on DFA State 1:

```
move ( { 1 2 3 5 6 8 9 10 11 } ,a )= { 2 10 }
e_closure ( { 2 10 })= { 1 2 3 5 6 8 9 10 11 } ( same as DFA state 1)
move ( { 1 2 3 5 6 8 9 10 11 } ,b )= { 4 12 }
e_closure ( { 4 12 })= { 1 3 4 5 6 8 9 12 13 } ( DFA State 3 )
```

The transition table for these 2 transitions can be represented by:

```
dtran[1,'a'] = 1
dtran[1,'b'] = 3
```

Applying transitions of a and b on DFA State 2:

```
move ( { 1 3 4 5 6 8 9 } ,a )= { 2 10 }
e_closure ( { 2 10 })= { 1 2 3 5 6 8 9 10 11 }  ( same as DFA State 1 )
move ( { 1 3 4 5 6 8 9 } ,b )= { 4 }
e_closure ( { 4 })= { 1 3 4 5 6 8 9 }           ( same as DFA State 2 )
```

The transition table for these 2 transitions can be represented by:

```
dtran[2,'a'] = 1
dtran[2,'b'] = 2
```

Applying transitions of a and b on DFA State 3:

```
move ( { 1 3 4 5 6 8 9 12 13 } ,a )= { 2 10 }
e_closure ( { 2 10 })= { 1 2 3 5 6 8 9 10 11 }  ( same as DFA State 1 )
move ( { 1 3 4 5 6 8 9 12 13 } ,b )= { 4 14 }
e_closure ( { 4 14 })= { 1 3 4 5 6 8 9 14 }     ( DFA State 4 )
```

The DFA state 4 is an accepting DFA state since it contains NFA State 14, which is an accepting NFA state.

The transition table for these 2 transitions can be represented by:

```
dtran[3,'a'] = 1
dtran[3,'b'] = 4
```

Applying transitions of a and b on DFA State 4:

```
move ( { 1 3 4 5 6 8 9 14 } ,a )= { 2 10 }
e_closure ( { 2 10 })= { 1 2 3 5 6 8 9 10 11 }  ( same as DFA State 1 )
move ( { 1 3 4 5 6 8 9 14 } ,b )= { 4 }
e_closure ( { 4 })= { 1 3 4 5 6 8 9 }           ( same as DFA State 2 )
```

The transition table for these 2 transitions can be represented by:

```
dtran[4,'a'] = 1
dtran[4,'b'] = 2
```

Note that the transitions have been applied on all of the DFA states (1,2,3,4) that signals the end of the algorithm. The resultant DFA is shown in Fig. 2.16.

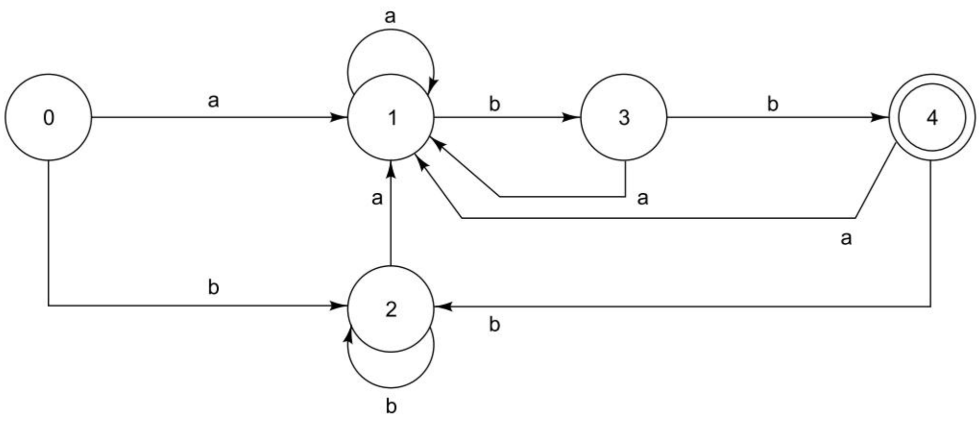


Fig. 2.16 *Resultant DFA for (a|b)*abb*

Algorithm 2.2 formalises the ideas that we just discussed.

```
start_state is the start state of the NFA
nfa_accept_set is the set containing all the accept states in the NFA
dfa_tab is the set of resulting dfa states after conversion of NFA to DFA
dtran is the Transition Table
```

```
nfa_to_dfa()
{
        e_closure_set = e_closure(start_state)
        Add a new DFA State to dfa_tab which corresponds to e_closure_set;
        If(e_closure_set ∩ nfa_accept_set){
             Mark the DFA State as an Accepting State
        }

        for(each DFA state T in dfa_tab ){
             for ( each possible input character 'c' )
                  current_set = The NFA Set corresponding to T;
                  move_set=move(current_set,c)
                  e_closure_set=e_closure(move_set)
                  if ( There is a DFA State d, which corresponds to e_closure_set
                  ){
                       dtran[T,c]= d ;
                  }else {
                       Add a new DFA State 'n' into dfa_tab which corresponds to
                       e_closure_set
                       If(e_closure_set ∩ nfa_accept_set){
                            Mark the DFA State as an Accepting State
                       }
                       dtran[T,c]=    n ;
                  }
             }
        }
}
```

Algorithm 2.2 *Conversion of NFA to DFA*

The DFA that we obtain by using Algorithm 2.2 can be used to determine if an input string matches a regular expression or not. We discuss the details on how the DFA can be interpreted to recognise a string in Section 2.3.2.2.

2.3.2.1 *Example 3—Converting NFA to DFA* This section demonstrates an example program that that converts an NFA to DFA using Algorithm 2.2. The program takes a regular expression on the command line, and prints out its NFA and the equivalent DFA. The NFA and DFA are shown in the form of a translation table. The dialog below shows the example program taking in different regular expressions and printing out the NFA and DFA details.

```
# Making the example
$ g++ -g -Wall rlex.cc ex3.cc -o ex3

# The NFA and the corresponding DFA for ab*c
$ ./ex3 'ab*c'

        NFA Transition Table

============ |==========================
State        | Input symbol
             |    a    b    c    epsilon
============ |==========================
START   0    |    -    -    -    { 1 }
        1    |    2    -    -    -
        2    |    -    -    -    { 5 }
        3    |    -    4    -    -
        4    |    -    -    -    { 6 3 }
        5    |    -    -    -    { 3 6 }
        6    |    -    -    -    { 7 }
        7    |  -   -   8   -
ACCEP   8    |  -   -   -   -
============ |==========================

       DFA Transition Table

============ |===============
State        |   Input symbol
             |   a   b   c
===========  |===============


START   0    |   1   -   -
        1    |   -   2   3
        2    |    -    2    3
ACCEP   3    |    -    -    -
============ |===============
# The NFA and the corresponding DFA for (a|b|c)+abb
$ ./ex3 '(a|b|c)+abb'

         NFA Transition Table
```

```
============ |==========================
State        |    Input symbol
             |    a    b    c    epsilon
============ |==========================
START   0    |    -    -    -    { 11 }
        1    |    2    -    -    -
        2    |    -    -    -    { 6 }
        3    |    -    4    -    -
        4    |    -    -    -    { 6 }
        5    |    -    -    -    { 1 3 }
        6    |    -    -    -    { 10 }
        7    |    -    -    8    -
        8    |    -    -    -    { 10 }
        9    |    -    -    -    { 5 7 }
        10   |    -    -    -    { 12 9 }
        11   |    -    -    -    { 9 }
        12   |    -    -    -    { 13 }
        13   |    14   -    -    -
        14   |    -    -    -    { 15 }
        15   |    -    16   -    -
        16   |    -    -    -    { 17 }
        17   |    -    18   -    -
ACCEP   18   |    -    -    -    -
============ |==========================

      DFA Transition Table

============ |===============
State        | Input symbol
             |    a    b    c
============ |===============
START   0    | 1  2    3
        1    | 4  2    3
        2    | 4  2    3
        3    | 4  2    3
        4    | 4  5    3
        5    | 4  6    3
ACCEP   6    |    4    2    3
============ |===============
# Verbose mode
$ ./ex3 -v 'ab'

        NFA Transition Table

============ |=====================
State        | Input symbol
             | a  b    epsilon
============ |=====================
START   0    | -  -    { 1 }
        1    | 2  -    -
        2    | -  -    { 3 }
```

```
       3     | -  4    -
ACCEP  4     | -  -    -
============ |=====================
DFA State=0 Corresponding NFA Set: { 0 1 }
DFA State=1 Corresponding NFA Set: { 2 3 }
DFA State=2 Corresponding NFA Set: { 4 }
   DFA Transition Table
============ |==========

State        | Input symbol
             |    a    b
============ |==========
START  0     | 1  -
       1     | -  2
ACCEP  2     |    -    -
============ |==========

# Another Regular Expression
$ ./ex3 -v '(a|b)+cd'

              NFA Transition Table

============ |===============================
State        |    Input symbol
             | a  b    c    d    epsilon
============ |===============================
START  0     |    -    -    -    -    { 7 }
       1     | 2  -    -    -    -
       2     | -  -    -    -    { 6 }
       3     | -  4    -    -    -
       4     | -  -    -    -    { 6 }
       5     | -  -    -    - { 1 3 }
       6     | -  -    -    - { 8 5 }
       7     | -  -    -    - { 5 }
       8     | -  -    -    - { 9 }
       9     | -  -    10 - -
       10    | -  -    -    - { 11 }
       11    | -  -    -    12   -
ACCEP  12    | -  -    -    -    -
============ |===============================

DFA State=0 Corresponding NFA Set: { 0 1 3 5 7 }
DFA State=1 Corresponding NFA Set: { 1 2 3 5 6 8 9 }
DFA State=2 Corresponding NFA Set: { 1 3 4 5 6 8 9 }
DFA State=3 Corresponding NFA Set: { 10 11 }
DFA State=4 Corresponding NFA Set: { 12 }

     DFA Transition Table

============ |====================
State        |    Input symbol
```

```
               | a  b    c    d
============ |====================
START   0    |    1    2    -    -
        1    |    1    2    3    -
        2    | 1  2    3    -
        3    | -  -    -    4
ACCEP   4    | -  -    -    -
============ |====================

# Another Regular Expression
$ ./ex3 -v '(a|b)*abb'

       NFA Transition Table

============ |======================
State        |    Input symbol
             |    a    b    epsilon
============ |=====================
START   0    |    -    -    { 7 }
        1    | 2  -    -
        2    | -  -    { 6 }
        3    | -  4    -
        4    | -  -    { 6 }
        5    | -  -    { 1 3 }
        6    | -  -    { 8 5 }
        7    | -  -    { 5 8 }
        8    | -  -    { 9 }
        9    | 10      -    -
        10   | -  -    { 11 }
        11   | -  12   -
        12   | -  -    { 13 }
        13   | -  14   -
ACCEP   14   | -  -    -
============ |=====================

DFA State=0 Corresponding NFA Set: { 0 1 3 5 7 8 9 }
DFA State=1 Corresponding NFA Set: { 1 2 3 5 6 8 9 10 11 }
DFA State=2 Corresponding NFA Set: { 1 3 4 5 6 8 9 }
DFA State=3 Corresponding NFA Set: { 1 3 4 5 6 8 9 12 13 }
DFA State=4 Corresponding NFA Set: { 1 3 4 5 6 8 9 14 }

     DFA Transition Table

============ |=================
State        |    Input symbol
             |    a    b
============ |=================
START   0    |    1    2
        1    |    1    3
        2    |    1    2
        3    |    1    4
ACCEP   4    |    1    2
============ |=================
```

2.3.2.2 ***Interpreting DFA*** In the last few sections, we learnt how to convert a regular expression to an NFA and create a DFA from an NFA. This section shows how a DFA can be interpreted to recognise the strings matching the regular expression.

Consider the regular expression ***'ab*c'*** for discussion. The NFA and the corresponding DFA can be obtained by using the executable *ex3* shown in Section 2.3.2.1.

```
# The NFA and the corresponding DFA for ab*c
$ ./ex3 'ab*c'

          NFA Transition Table

===========|==========================
State      | Input symbol
           | a     b   c   epsilon
===========|==========================
START   0  | -     -   -   { 1 }
        1  | 2     -   -   -
        2  | -     -   -   { 5 }
        3  | -     4   -   -
        4  | -     -   -   { 6 3 }
        5  | -     -   -   { 3 6 }
        6  | -     -   -   { 7 }
        7  | -     -   8   -
ACCEP   8  | -     -   -   -
===========|==========================

     DFA Transition Table

===========|===============
State      | Input symbol
           | a     b   c
===========|===============
START   0  | 1     -   -
        1  | -     2   3
        2  | -     2   3
ACCEP   3  | -     -   -
===========|===============
```

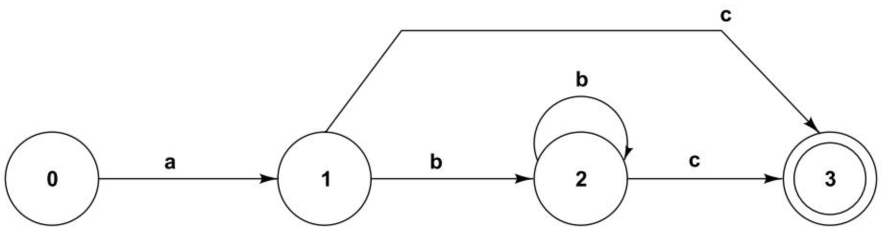


Fig. 2.17 *DFA for 'ab*c'*

Figure 2.17 shows the visual representation of the DFA for ***'ab*c'*** derived above by running Example 3.

Let us simulate an input string '**abbbc**' to the above DFA. The initial state of the DFA is state 0. The following table shows the transitions occurring due to the input string.

Current state	Input character	Next state
0	a	1
1	b	2
2	b	2
2	b	2
3	c	2 (accepting state)

The transitions above yield an accepting state (state 3) and hence the string is *recognised to match the RE*. Lets consider another input **'abbabbc',** for which the transitions are shown below:

Current state	Input character	Next state
0	a	1
1	b	2
2	b	2
2	a	No transition specified. So retry the transition from the start state
0	a	1
1	b	2
2	b	2
2	c	2 (accepting state)

Again, the transitions above yield an accepting state (state 3) and hence the string is *recognised to match the RE*.

The transitions occurring in the DFA can be formalised in the following algorithm:

```
cur_state holds the state number
move(state,c) gives the next state by looking at the Transition Table
```

```
interpret_dfa()
{
        cur_state=0;
        c=nextchar();

        while ( c is not end-of-input ) {
             cur_state = move (cur_state, c);
             if ( cur_state is ACCEPTING_STATE ) {
                  return (SUCCESS);
             }
             c = nextchar();
        }
        return(FAILURE);
}
```

Algorithm 2.3 *DFA interpretation*

2.3.2.3 *Example 4—Interpreting DFA* This section demonstrates an example program that is similar to the functionality of 'egrep' utility. The program uses the DFA to recognise if the input matches the regular expression or not. The program takes a regular expression and file name to search for a string that matches the regular expression. The output of the program is the lines matching the regular expression along with the line numbers. The dialog below shows the example program taking in different regular expressions and file names, and printing out the lines in the file matching the regular expression.

```
# Making the example
$ g++ -g -Wall rlex.cc ex4.cc -o ex4
# Use the file that we used before for learning about the Regular Expressions
$ cat -n my_input
    1  sink
    2  base
    3  start
    4  dog
    5  goat
    6  boost
    7  easter
    8  ported
    9  global
   10  boss
   11  labs

# Matches b followed by zero or any number of o then s
$ ./ex4 'bo*s' my_input
6:boost
10:boss
11:labs

# Matches b followed by one or any number of o then s
$ ./ex4 'bo+s' my_input
6:boost
10:boss

# Matches b followed by one or no o then s
$ ./ex4 'bo?s' my_input
10:boss
11:labs

# Matches oa or ort
$ ./ex4 'oa|ort' my_input
5:goat
8:ported

# Matches oa or os followed by t
$ ./ex4 '(oa|os)t' my_input
5:goat
6:boost

# Verbose mode
# Test file containing strings
$ cat -n test5
    1  abbbc
    2  cbbbc
    3  aaaaaaaabbbc

$ ./ex4 -v '(a|b)*abb' test5

       NFA Transition Table

==========|=====================
State     |  Input symbol
          |  a   b    epsilon
==========|=====================
START   0 |  -   -   { 7 }
        1 |  2   -   -
        2 |  -   -   { 6 }
        3 |  -   4   -
        4 |  -   -   { 6 }
        5 |  -   -   { 1 3 }
        6 |  -   -   { 8 5 }
        7 |  -   -   { 5 8 }
        8 |  -   -   { 9 }
        9 |  10  -   -
       10 |  -   -   { 11 }
       11 |  -   12  -
       12 |  -   -   { 13 }
       13 |  -   14  -
ACCEP  14 |  -   -   -
==========|=====================
     DFA Transition Table
==========|================
State     | Input  symbol
          |  a     b
==========|================
START   0 |  1     2
        1 |  1     3
        2 |  1     2
        3 |  1     4
ACCEP   4 |  1     2
==========|================
1:abbbc
3:aaaaaaaabbbc
```

2.3.3 Minimisation of DFA States

In Section 2.3.2.3, we learnt about converting an NFA to DFA and how DFA can be interpreted to recognise strings that match a regular expression. When an NFA is converted into DFA, the resulting number of DFA states is not optimised. It is possible to reduce the number of DFA states, thereby consuming lesser memory for storing the DFA data structures and improving memory efficiency. This section discusses the techniques to reduce the number of DFA states to a minimum for a given DFA.

From the lexical analyser generator development point of view, this is the third step (minimisation of DFA states) shown in Fig. 2.10.

Consider the regular expression ***'(a|b|c)+abb'***. We can get the DFA by invoking the binary created as a part of Example 3, as shown in the dialog below.

```
# The NFA and the corresponding DFA for (a|b|c)+abb
$ ./ex3 '(a|b|c)+abb'

         NFA Transition Table

=========  |==========================
State      |   Input symbol
           |   a     b     c   epsilon
============|==========================
START  0   |   -     -     -     { 11 }
       1   |   2     -     -     -
       2   |   -     -     -     { 6 }
       3   |   -     4     -     -
       4   |   -     -     -     { 6 }
       5   |   -     -     -     { 1 3 }
       6   |   -     -     -     { 10 }
       7   |   -     -     8     -
       8   |   -     -     -     { 10 }
       9   |   -     -     -     { 5 7 }
      10   |   -     -     -     { 12 9 }
      11   |   -     -     -     { 9 }
      12   |   -     -     -     { 13 }
      13   |   14    -     -     -
      14   |    -    -     -     { 15 }
      15   |   -     16    -     -
      16   |   -     -     -     { 17 }
      17   |   -     18    -     -
ACCEP 18   |   -     -     -     -
==========|==========================

    DFA Transition Table

==========|===============
State      | Input   symbol
           |   a     b     c
==========|===============
START  0   |   1     2     3
       1   |   4     2     3
       2   |   4     2     3
       3   |   4     2     3
       4   |   4     5     3
       5   |   4     6     3
ACCEP  6   |   4     2     3
==========|===============
```

The following observations can be made by looking at the DFA states 1, 2, and 3.

- All the 3 states make a transition to DFA state 4, given the input ‘a’.
- All the 3 states make a transition to DFA state 2, given the input ‘b’.
- All the 3 states make a transition to DFA state 3, given the input ‘c’.
- The transitions for states 1, 2 and 3 are defined for inputs a, b, c only.

We can replace the DFA states 1, 2 and 3 by a single state, which has transitions to itself on inputs b and c. On given an input of ‘a’, the transition will be to the DFA state 4. It is likely that the state 4 might have to be re-named to different state (for example, state 2 or 3) in the wake of reduction of three states to a single state.

Comparing state 4 and state 5 in the above DFA, both of the states make a transition to state 4 on receiving an input ‘a’; both of the states make a transition to 3 on receiving input of ‘c’. However, on receiving the input ‘b’ state 4 transitions to 5, while state 5 transitions to 6. The states 4 and 5 are said to be ***distinguished*** from each other by the input ‘b’.

The algorithm for reducing the number of DFA states works by:

- Creating groups of states that are distinguished by some input string.
- Merging the group of states that cannot be distinguished into a single state.

Let us try to minimise the DFA states in the example by using the above principle. In the first pass, we scan all the DFA states to separate out the accepting states and non-accepting states, this would yield 2 groups as shown in the table below:

DFA states	Group
0,1,2,3,4,5 (non-accepting)	1
6 (accepting)	2

In the second pass, we compare the states in each of the groups to separate out the states that are not identical. Thus, we compare state 0 with state 1, with respect to transitions on all possible inputs, we find that in state 0, the transition for ‘a’ is 1, while in state 1, the transition for ‘a’ is 4. Hence, we create a new group 3, into which state 1 is placed. Next, we compare state 0 with 2, they are different with respect to transitions on input ‘b’, hence state 2 is also moved to group 3. This kind of comparison of state 0 is done with 3, 4, and 5 also, since none of them are identical to 0 with respect to transitions on all inputs, all of them are moved to group 3. Thus we have at the end of the comparisons,

DFA states	Group
0	1
6 (accepting)	2
1,2,3,4,5	3

Next, we perform the same kind of operation on group 3. We compare state 1 with state 2 with regard to transitions on all possible inputs. Since they are identical, we retain the state 2 in group 3 itself. The same happens for state 3. When state 1 is compared with state 4, it is found that they are different with respect to transition on ‘b’, hence it will be moved into a new group 4. When state 1 is compared with 5, it is found that they are different with respect to transition on ‘b’, hence it will be moved into group 4. Thus we have at the end of the comparisons,

DFA states	Group
0	1
6 (accepting)	2
1,2,3	3
4,5	4

Next, we perform the same kind of operation on group 4. We compare state 4 with state 5 with regard to transitions on all possible inputs. When state 4 is compared with state 5, it is found that they are different with respect to transition on ‘b’, hence state 5 will be moved into a new group 5. Thus we have at the end of the comparisons,

DFA states	Group
0	1
6 (accepting)	2
1,2,3	3
4	4
5	5

Since group 5 contains only 1 element, there would be no comparisons necessary.

We have successfully created 5 groups of states. Note that the elements of each group have identical transactions with respect to all inputs and can be merged into a single state. We can assign new state numbers to the groups as shown in the following table :

DFA states	Group	New DFA state
0	1	0
6 (accepting)	2	1
1,2,3	3	2
4	4	3
5	5	4

Thus we were able to reduce the number of DFA states from 7 to 5 by creating groups of states that are distinguished by some input string and merging the group of states that cannot be distinguished into a single state. This algorithm has been formalised below:

```
dfa_tab is a array containing all the DFA States
Groups is an array of group
```

```
minimise_dfa ()
{
        for ( each group g in Groups ){
          d1 = first DFA State in the Group g ;
          new_group is empty;
          for ( each subsequent DFA State d in g ){
              if ( DFA State d is not identical to d1 with respect
                  to transitions on all possible inputs ){
                Add it to new_group
           }
      }
      if(new_group is not empty)
           Add new_group to Groups
      }
   }
   Create a new Transitions Table with group index as the state number
}
```

Algorithm 2.4 *Minimise DFA*

2.3.3.1 ***Example 5—Minimising DFA States*** This section demonstrates an example program that can translate a given regular expression into NFA, convert the NFA to DFA and minimise the DFA states. The program implements Algorithm 2.4 for minimising the DFA states. The program takes a regular expression on the command line, translates it into NFA, converts the NFA to DFA, minimises the DFA states and prints out the NFA/DFA translation tables. The dialog below shows the example program taking in different regular expressions and printing out the NFA and DFA details.

```
# Making the example
$ g++ -g -Wall rlex.cc ex5.cc -o ex5

# Verbose mode
# Minimising DFA States
$ ./ex5 -v '(a|b)*abb'

      NFA Transition Table

============ |=====================
State        |    Input  symbol
             |    a      b     epsilon
============ |=====================
START  0     |    -      -     { 7 }
       1     |    2      -     -
       2     |    -      -     { 6 }
       3     |    -      4     -
       4     |    -      -     { 6 }
       5     |    -      -     { 1 3 }
       6     |    -      -     { 8 5 }
       7     |    -      -     { 5 8 }
       8     |    -      -     { 9 }
       9     |    10     -     -
      10     |    -      -     { 11 }
      11     |    -      12    -
      12     |    -      -     { 13 }
      13     |    -      14    -
ACCEP 14     |    -      -     -
============ |=====================

    DFA Transition Table
```

```
============ |================
State        |    Input symbol
             |    a       b
============ |================
START  0     |    1       2
       1     |    1       3
       2     |    1       2
       3     |    1       4
ACCEP  4     |    1       2
============ |================
No of Old States=5 No of New States=4
Old State=0 New State=0
Old State=1 New State=2
Old State=2 New State=0
Old State=3 New State=3
Old State=4 New State=1

    MIN DFA Transition Table

============ |================
State        |    Input symbol
             |    a       b
============ |================
START  0     |    2       -
ACCEP  1     |    2       -
       2     |    2       3
       3     |    2       1
============ |================
# Another Regular Expression
$ ./ex5 -v '(a|b|c)+'

            NFA Transition Table

============ |==========================
State        |    Input symbol
             |    a       b     c     epsilon
============ |==========================
START  0     |    -       -     -     { 11 }
       1     |    2       -     -     -
       2     |    -       -     -     { 6 }
       3     |    -       4     -     -
       4     |    -       -     -     { 6 }
       5     |    -       -     -     { 1 3 }
       6     |    -       -     -     { 10 }
       7     |    -       -     8     -
       8     |    -       -     -     { 10 }
       9     |    -       -     -     { 5 7 }
      10     |    -       -     -     { 12 9 }
      11     |    -       -     -     { 9 }
ACCEP 12     |    -       -     -     -
=============|==========================
      DFA Transition Table

============ |===============
State        |    Input symbol
             |    a      b     c
============ |======================
START  0     |    1      2     3
ACCEP  1     |    1      2     3
ACCEP  2     |    1      2     3
ACCEP  3     |    1      2     3
============ |======================

No of Old States=4 No of New States=2
Old State=0 New State=0
Old State=1 New State=1
Old State=2 New State=1
Old State=3 New State=1

    MIN DFA Transition Table

============ |===================
State        |    Input symbol
             |    a      b     c
============ |===================
START  0     |    1      1     1
ACCEP  1     |    1      1     1
============ |===================
```

2.3.4 Generate the Code for the Lexical Analyser using the Minimised DFA

In the previous sections we understood the concepts of converting a regular expression into NFA, converting the NFA to DFA and then minimising the number of DFA states. Now, we enter the last phase of lexical analysis (step 4), namely the generation of C language code that can be compiled to make a lexical analyser.

As we saw in Section 2.2.1, the main data structures that are required to interpret a DFA are (1) transition table; (2) flag to indicate whether a given state is accepting state or not. The DFA interpreting algorithm (Algorithm 2.3) operates on these data structures to determine if a given input matches the regular expression or not. The DFA interpreting algorithm is a standard one and hence the code for it can be generated statically from a template. The two data structures mentioned above are the only thing that varies depending on the input regular expression. Thus in order to generate a lexical analyser, we need to:

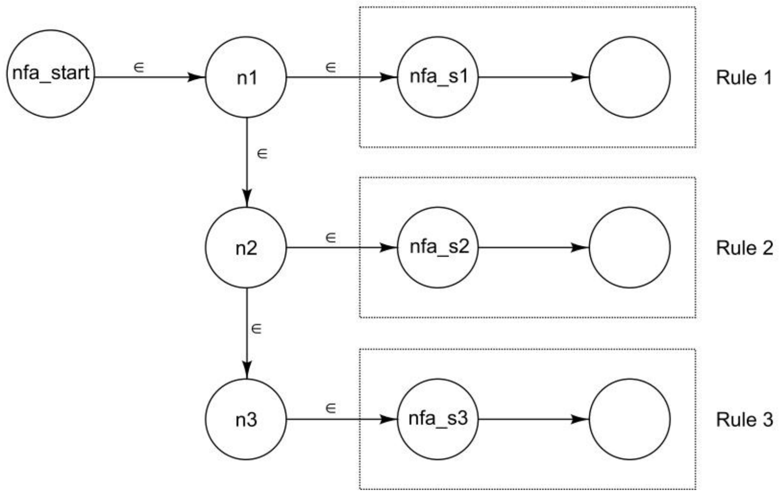
(a) Export the transition table and an accept marker that can indicate if a given state is an accepting state or not into the lexical analyser. These two data structures are dependent on the input lexical specification file.

(b) Generate the code for a standard DFA interpreting algorithm (Algorithm 2.3) that remains the same regardless of the input lex file. This is the recogniser component of lexical analyser shown in Fig. 2.5.

(c) Generate the code for the action component shown in Fig. 2.5. This is dependent on the input lexical specifications file.

The following describes the design issues that need to be addressed for generating lexical analyser code

- In a lexical analyser, even when the input character string does not match the regular expression, there is usually some default action that needs to be taken, like echoing the character onto the screen. This can be taken care by having a stack in which input characters are stored and then control can flow to the specific action block or the default action block depending on whether it matches a regular expression or not.
- For generating the code of standard DFA interpretation algorithm, there are two possible approaches: (a) generate the code from a template file or (b) generate the code from stored static strings in the lexical analyser generator. In *rlex*, the latter approach has been chosen.
- In this chapter, till now, we were having a single regular expression to convert to NFA then to DFA, and interpret it. In a lexical specification file, there could be a number of rules (regular expressions), which need to be addressed simultaneously. Consider an input lexical specification file having 3 rules. After converting them to NFA individually, each of them would have a start state say nfa_s1, nfa_s2 and nfa_s3. A single NFA can be built out of them by creating 3 new NFA nodes (n1, n2, n3) and interconnecting these using the ∈ transitions**.** A new NFA start node (nfa_start) can be used as a start for the entire NFA. This is shown in the figure below. This NFA can then be converted to DFA and then interpreted.



- The interpretation of DFA shown in Algorithm 2.3 returns with the shortest sub-string that matches a regular expression, but a lexical analyser needs to return the longest sub-string that matches the regular expression. This type of regular expression matching where the longest sub-string match is returned is called greedy interpretation of the RE; e.g. consider a regular expression '*abc**' and an input string *'abccccd',* In a non-greedy interpretation, the RE match would yield ab while the greedy interpretation would return *'abcccc'*. In order to cater to greedy version, the DFA Interpretation shown in Algorithm 2.3, needs to be modified a little bit as shown in Algorithm 2.5. The modification is to keep track of the last accepting state and continue taking the input until there is transition from accepting state to non-accepting state. When such a transition occurs, we signal that the match is complete. In lexical analyser generator *rlex*, we generate code for lexical analyser which uses the greedy interpretation of DFA shown in Algorithm 2.5.

```
cur_state stores the current state of the DFA
move(state,c) gives the next state by looking at the Transition Table
```

```
interpret_greedy_dfa()
{
        last_accept_state = -1
        while ( c is not end-of-input ) {
             cur_state = move(cur_state, c);
             if ( cur_state is ACCEPTING_STATE ) {
                  last_accept_state=cur_state;
             }else{
                  if(last_accept_state != -1 ){
                       return(SUCCESS);
                  }
             }
             c = nextchar();
        }
        if(last_accept_state != -1){
                  return(SUCCESS);
        } else {
                  return(FAILURE);
        }
}
```

Algorithm 2.5 *Greedy DFA interpretation*

2.3.4.1 ***Example 6—Lexical Analyser Code Generation using the minimised DFA*** This section demonstrates a lexical analyser generator—rlex, which generates code for a lexical analyser given a lexical specifications file. 'rlex' is a lexical analyser generator similar to 'flex'. It implements most of the theory we have learnt in the previous sections. 'rlex' takes in a lexical specifications file and generates code for a corresponding lexical analyser. The 'rlex' lexical analyser generator handles only a few of the meta characters shown in Table 2.1, hence it cannot work with full-fledged C language lexical specifications. The dialog below shows 'rlex' taking in 'keywords.l'—a lexical specification that identifies and returns only the keywords as tokens and ignores everything else. The dialog illustrates rlex lexical analyser generator taking the input lexical specifications and generating code for the lexical analyser.

```
# Building the lexical Analyzer Generator
$ g++ -g -Wall rlex.cc ex6.cc -o rlex

# Compiling Lex file using rlex to generate C Code for Lexical Analyzer
$ ./rlex keywords.l >out.c

# Building Lexical Analyzer Binary
$ gcc -g out.c -o lexer

# Input file to be tokenized
$ cat -n test8.c
    1  int
    2  main ()
```

```
{
    int i = 0;

    while (i < 5) {
        puts ("Hi\n");
        i++;
    }

    for (i = 0; i < 10; i++) {
        puts ("Hello World\n");
    }
}

# Lexical Analyser at work !
$ ./lexer test8.c
Lexeme=[int]        Length=3    Token is INT
Lexeme=[int]        Length=3    Token is INT
Lexeme=[while]      Length=5    Token is WHILE
Lexeme=[for]        Length=3    Token is FOR
```

SUMMARY

Lexical analysis is the first stage in compilation of a source program written in a higher-level language like C or C++. The lexical analyser reads the input source program and produces as output a sequence of tokens that the parser uses for syntax analysis. In order to facilitate the development of lexical analysers for any higher-level language easily, lexical analyser generators are used. A lexical analyser generator is a tool that can generate code to perform lexical analysis of the input, given the rules for the basic building blocks of the language. The rules for the basic building blocks of a language are called its lexical specifications. This chapter describes in detail the 4 steps that lexical analyser generators follow in order to generate the lexical analyser from the lexical specifications.

REVIEW QUESTIONS AND EXERCISES

2.1 What are the main functions of a lexical analyser?

2.2 State whether the following statements are true or false:
- (a) A lexical analyser strips out the comments and white spaces from input source program.
- (b) The part of the input stream that qualifies for a certain type of token in a lexical analyser is called as lexeme.
- (c) The lexical analyser keeps track of the new line characters, so that it can output the line number with associated error messages, in case of errors in the input source program.
- (d) The lexical analyser helps in creation of symbol table.

2.3 Why do we need to have lexical analyser generators? What are its advantages?

2.4 How does a lexical analyser generator like 'flex' convert the lexical analysis specification to a lexical analyser? What are the steps involved?

2.5 What is a regular expression? How is it used in lexical specifications?

2.6 What is a regular definition? Specify an identifier and a hexadecimal constant in C language taking the help of regular definitions.

2.7 What are the limitations of regular expressions? Can a regular expression be used for detecting simple C language statements?

2.8 What are the various components of a lexical specification file? Illustrate with an example.

2.9 What is a finite state machine? How is it visually represented? Give an example of FSM to recognise 3 keywords in C language?

2.10 What are deterministic finite machines and non-deterministic finite machines? List out the differences between them.

2.11 How do you translate a regular expression to an NFA? Translate (a|b|c)?d into an NFA and draw its transition diagram.

2.12 Write an algorithm to interpret an NFA and report if a given string matches the NFA. Illustrate with an example.

2.13 What is a move set? What is an epsilon closure set? Give examples. How are they used in reporting if a given string matches a regular expression?

2.14 Why do lexical analyser generators use and interpret DFA to report if a given input string matches the given regular expression, rather than using an NFA? What are the advantages of using a DFA?

2.15 How is an NFA converted to a DFA? Illustrate for a sample regular expression '(a|b)*abb'.

2.16 Write an algorithm to interpret a DFA and report if a given string matches the DFA.

2.17 Why is the step to minimise the number DFA states required during the process of generating a lexical analyser from the lexical specifications? Give the algorithm to minimise the number of DFA states for a given DFA.

2.18 What are the main design issues to be considered while generating the code for lexical analyser given the minimised DFA?

2.19 State whether the following statements are TRUE or FALSE:
   (a) A lexical analyser generator translates the regular expressions given in lexical specification file into NFA. The NFA is then converted to DFA. The DFA is minimised to eliminate duplicate states. The minimised DFA is then used to generate the lexical analyser.
   (b) An NFA cannot be used to recognise whether a given input string matches it or not. Hence, the NFA is converted to DFA in a lexical analyser generator.
   (c) The generated lexical analysers use a greedy interpretation while matching the regular expressions provided in the lexical specifications file.
   (d) A lexical analyser generator cannot be used to generate lexical analyser for different programming languages.

2.20 The extended regular expressions offer more facilities to recognise patterns than the basic regular expressions. List out the additional meta characters and their significance in the extended regular expressions.

2.21 Most of the operating systems provide a regular expression library that can be used in the programs to match input strings. Write a program on LINUX to use the 'regex' library to check if a given string is a valid internet protocol address (IP address)?

2.22 Write a regular expression that can detect C language comments. Support both the single line comment syntax (//) and multi-line comment syntax (/* */).

2.23 A meta-character is used to signify a special meaning within a regular expression. How do you turn off the special meaning and recognise a string containing the meta character literally? Write a regular expression that can detect sentences in a text file containing a full-stop at the end of the line.

2.24 Write a lexical analyser that can recognise an identifier and a few of the keywords in C language without using a lexical analyser generator. Compare the effort required for implementing the same by having a lexical analyser generated from a lexical specification file. This should justify the need to have lexical analyser generator tools.

2.25 Write a lexical specifications file supporting a small subset of ANSI-C language and generate the lexical analyser for the same using 'flex'. Check whether the lexical analyser can tokenise some sample input C files.

SYNTAX ANALYSIS

3

Introduction

Syntax analysis is the second stage in compilation of a source program after lexical analysis. While lexical analyser reads the input source program and produces as output a sequence of tokens, syntax analysis verifies if the tokens are properly sequenced in accordance with the grammar of the language. Consider, for example a C program as input to the compiler as shown in Fig. 3.1. To understand the function of syntax analysis, let us take the statement in line number 8 of the input C program. The lexical analyser separates it into 5 tokens, namely, identifier (count), equal to operator (=), identifier (count), plus operator (=), constant (1), followed by semicolon (;). The syntax analyser verifies if the sequence of identifier followed by equal to operator, identifier, plus operator, and a constant is a valid sequence in the grammar of C language. The syntax analyser returns success, if the sequence is defined in the grammar of C language, or failure in case it is not defined. In cases where the statement does not match the grammar specified for the language, the syntax analyser detects the error, emits appropriate error message to the user and if possible, recovers from the error.

The syntax analysis (or ***parsing***) is performed by a module in the compiler called as ***syntax analyser*** or ***parser***. For the parser to perform syntax analysis, the grammar of the language needs to be specified. How do we specify the grammar of the language? ***Context-free grammar*** (CFG) is usually used to define the grammar of a language. Context-free grammar can be adequately used for describing any programming language.

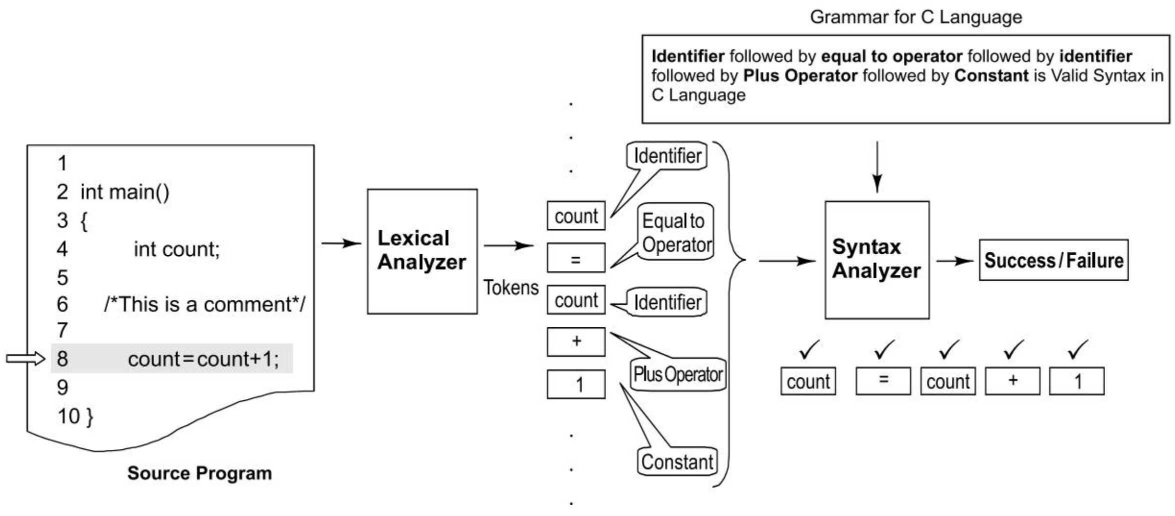


Fig. 3.1 *Syntax analysis*

Section 3.1 deals with how we can specify the context-free grammar using some standard notation. Section 3.2 shows the classification of various techniques used for parsing. We discuss one class of parsing techniques called as top-down parsing in Section 3.4. Section 3.5 is devoted to another class of parsing techniques called as bottom-up parsing. Apart from verifying if the tokens are properly sequenced in accordance with the grammar of the language, the syntax analyser is responsible for reporting errors in the input source program and recovering from them. This is dealt with in Section 3.3.

3.1 CONTEXT-FREE GRAMMAR

Context-free grammar (CFG) is used to define the syntactic structure of a programming language. It contains a set of rules called as ***productions*** or ***production rules***.

To understand the productions and the other aspects of context-free grammar, let us take a look at a grammar that can describe a simple assignment statement in C language, like any one of the following:

```
count = 5 ;                    /* Variant 1 */
count = index ;                /* Variant 2 */
count = 3*2 ;                  /* Variant 3 */
count = count + index ;        /* Variant 4*/
count = count + 1 ;             /* Variant 5 */
count = count * 2 + 5 ;         /* Variant 6 */
count = count * 2 + index ;     /* Variant 7 */
```

The context-free grammar in Table 3.1 defines the syntax of a simple assignment statement in C language such as the ones shown above. The context-free grammar in Table 3.1 contains 4 productions numbered accordingly.

Table 3.1 *Context-free grammar*

1	c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression → CONSTANT
3	c_expression → IDENTIFIER
4	c_expression → c_expression OPERATOR c_expression

The → in each of the productions of Table 3.1 may be read as 'can take the form'. The production rule 1 can thus be interpreted as 'A C statement can take the form of an IDENTIFIER followed by an equal-to-operator, C expression and a semicolon'. The production 2 can be interpreted as, 'A C expression can take the form of a CONSTANT'. The production 3 can be interpreted as 'A C expression can take the form of an IDENTIFIER'. The production 4 can be interpreted as 'A C expression can take the form of a C expression followed by an OPERATOR and a C expression'. The production 4 defines the C expression in terms C expression itself recursively.

In general, a context-free grammar consists of the following components:

1. A set of tokens called as ***terminal symbols***. These tokens cannot be sub-divided into smaller elements. The lexical analyser provides these as tokens as a part of tokenising the input. In the above example, IDENTIFIER, CONSTANT, SEMI_COLON, OPERATOR and EQ_TO_OP are the terminal symbols. They are denoted by capital letters. In the above example, the lexical analyser returns OPERATOR as the token type on encountering any of the operators '+' or '–' or '*' or '/'.
2. A set of ***non-terminal symbols***. These can be broken down into smaller components. In the above example, c_expression, and c_statement are the non-terminal symbols. These are usually denoted by lower-case letters. The recursive definition of non-terminals (e.g. production 4) is extensively used while defining grammar for programming languages.
3. A set of ***productions*** where each production consists of a non-terminal called as the left-side of the production, an arrow and a sequence of tokens and/or non-terminals called as the right-side of the production. For example, the c_expression is the left-side and CONSTANT is the right-side of production 2.
4. A designation of one of the non-terminals called as ***start symbol***. This requires some explanation. Let's say we want to check whether the input is a simple C assignment statement such as variant 1 or variant 2. In such case, the start symbol is c_statement. Let's say in another scenario, we want to check whether the input is a simple C expression. In such case, the start symbol is c_expression. In the above example, since we are looking to validate one simple C assignment statement as input, the start symbol is a c_statement.

The grammar of a programming language expressed in the form shown in of Table 3.1 is also referred to as ***Backus-Naur form (BNF)*** in recognition to the contributions by the authors Backus and Naur. There are several alternatives to the notation used to represent a grammar in BNF. Table 3.2 shows one of the popular notations accepted by automatic parser generators like bison and yacc. The notation used in Table 3.2 uses the colon (:) symbol instead of the → symbol to separate the LHS and the RHS of a production. The semicolon (;) is used to indicate the end of the production.

Table 3.2 *Production rules for a single C-statement using an alternate notation*

1	c_statement	:	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
		;	
2	c_expression	:	CONSTANT
		;	
3	c_expression	:	IDENTIFIER
		;	
4	c_expression	:	c_expression OPERATOR c_expression
		;	

Apart from these the symbol of vertical bar (|) indicating an OR condition is also used commonly to show the production rules having the same LHS together. The production rules 2, 3 and 4 can be shown together using '|' as shown below:

2	c_expression	:	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR c_expression
		;	

By looking at production rules 2, 3, 4 we can infer that c_expression consists of CONSTANT (rule 2) *or* IDENTIFIER (rule 3) *or* c_expression followed by OPERATOR and c_expression (rule 4).

Till now the discussion was mostly with regard to defining and expressing context-free grammar. Let us now see how to identify if a given input string is in conformance to the syntax specified by the grammar or not.

In order to prove that an input is syntactically in conformance to grammar, we need to prove that it can take one of the generic forms deduced from the productions. There are several ways of proving this. One of the methods involves the following procedure.

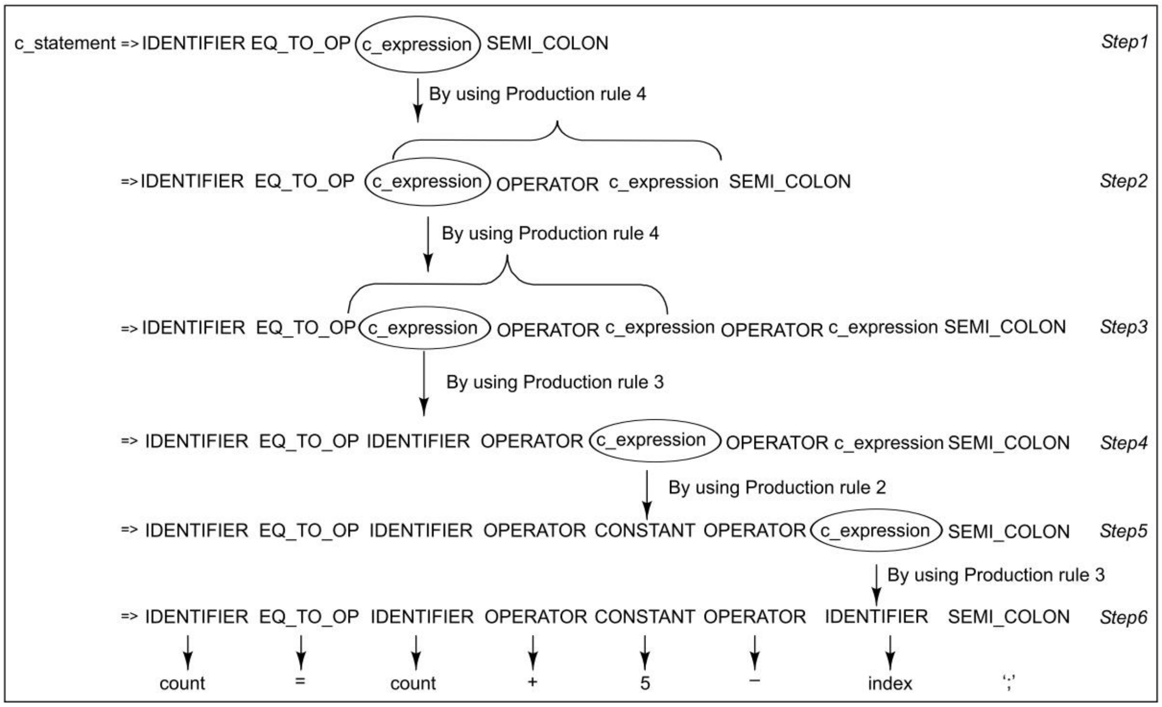
1. Expand the start symbol. For example, if we are trying to prove that an input is a C statement conforming to the grammar specified in Table 3.1, the start symbol is the c_statement.
2. Repeat step 3 until there are no more non-terminals to replace.
3. Replace the ***leftmost*** non-terminal by one of its right-hand sides.
4. Exit.

This process of showing how an input can be verified for conformance to a grammar is called as ***derivation***. Figure 3.2 illustrates the method for checking if the input *'count = count + 5 – index;'* is in conformance with the grammar in Table 3.1. We use a symbol **=>,** which means 'derives' in the proof.

Figure 3.2 proves that the statement *count = count + 5 – index;* is indeed part of the grammar specified in Table 3.1.

The entire sequence of replacements as seen in Figure 3.2 is called as derivation of *'count = count + 5 – index;'* from c_statement. It shows the proof that one instance of c_statement is *'count = count + 5 – index;'*.

The process used in Figure 3.2 for the derivation of *'count = count +5 – index;'* is called as ***leftmost derivation*** because we replaced the leftmost non-terminal in the partially parsed sentence with equivalent production's right-hand side.



c_statement => IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
Step1
By using Production rule 4
=> IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression SEMI_COLON
Step2
By using Production rule 4
=> IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression OPERATOR c_expression SEMI_COLON
Step3
By using Production rule 3
=> IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR c_expression OPERATOR c_expression SEMI_COLON
Step4
By using Production rule 2
=> IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR CONSTANT OPERATOR c_expression SEMI_COLON
Step5
By using Production rule 3
=> IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR CONSTANT OPERATOR IDENTIFIER SEMI_COLON
Step6
count
=
count
+
5
–
index
';'

Fig. 3.2 *Derivation of 'count= count +5 –index ;'*

In step 1, we saw that the c_statement (start symbol of the grammar in Table 3.2) took a form 'IDENTIFIER EQ_TO_OP c_expression SEMI_COLON'. In step 2, we saw that the c_statement took a form 'IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression SEMI_COLON'. In all, we saw 6 forms corresponding to each step in Fig. 3.2. These forms are called ***sentential forms*** of the grammar. In the final step (step 6), the form is 'IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR CONSTANT OPERATOR IDENTIFIER SEMI_COLON. 'All the elements of this form are terminals. This is called as a ***sentence*** of the grammar in Table 3.2. In short a sentence is a sentential form in which all the elements are terminals only. Using a different substitution for c_expression, you can prove that the form 'IDENTIFIER EQ_TO_OP IDENTIFIER SEMI_COLON' is another sentence generated out of the same grammar in Table 3.2. The collection of all the sentences that can be derived from the grammar is called as ***context-free language.*** If two grammars generate the same context-free language, then grammars are said to be equivalent.

In the above derivation of '*count* = *count* + *5* – *index*;', we replaced the leftmost non-terminal in the sentential forms with the production's right-hand side. Similarly, it is possible to replace the rightmost non-terminal in the partially parsed sentence with equivalent production's right-hand side and prove that the given statement is a C statement. This type of derivation is called as ***rightmost derivation*** or ***canonical derivation***. Shown below is the derivation of *count = count + 5 – index;* using the rightmost derivation. The comment at the right in italics at each of the steps below indicates the production used at that particular step in the derivation. For example, by P3 indicates using the production 3.

c_statement	=>	IDENTIFIER	EQ_TO_OP	c_expression					SEMI_COLON	*By P1*
	=>	IDENTIFIER	EQ TO_OP	c_expression	OPERATOR	c_expression			SEMI_COLON	*By P4*
	=>	IDENTIFIER	EQ TO_OP	c_expression	OPERATOR	c_expression	OPERATOR	c_expression	SEMI_COLON	*By P4*
	=>	IDENTIFIER	EQ TO_OP	c_expression	OPERATOR	c_expression	OPERATOR	IDENTIFIER	SEMI_COLON	*By P3*
	=>	IDENTIFIER	EQ TO_OP	c_expression	OPERATOR	CONSTANT	OPERATOR	IDENTIFIER	SEMI_COLON	*By P2*
	=>	IDENTIFIER	EQ TO_OP	IDENTIFIER	OPERATOR	CONSTANT	OPERATOR	IDENTIFIER	SEMI_COLON	*By P3*
	=>	count	=	count	+	5	–	index	;	

A ***parse tree*** is a graphical representation of a derivation. Let's see what a parse tree looks like and how it is constructed by taking the example of the derivation of a C statement *'count = count + 10 ;'* with the productions mentioned in Table 3.1.

The leftmost derivation proceeds as follows:

c_statement	=>	IDENTIFIER	EQ_TO_OP	c_expression				*By P1*
	=>	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR	c_expression	SEMI_COLON	*By P4*
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	c_expression	SEMI_COLON	*By P3*
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	CONSTANT	SEMI_COLON	*By P2*
	=>	count	=	count	+	10	;	

The construction of the parse tree for the derivation is shown in Fig. 3.3. The final parse tree in Fig. 3.3(e) can be used to represent the entire derivation. The numbering of the interior nodes specifies the order in which the derivation proceeded. Observe that the numbering of the interior nodes in the parse tree is actually a pre-order numbering. To recall, a pre-order traversal of tree visits the root, followed by left sub-tree and then the right sub-tree. To sum up, a pre-order traversal of the interior nodes of the final parse tree replicates the entire leftmost derivation in totality.

A parse tree has the following properties:

- The root node is labelled with the start symbol. In the parse tree example shown at Fig. 3.3(e) the root node is c_statement.
- Each of the interior nodes, i.e. nodes having children, are labelled by a non-terminal. At each of the interior nodes, the children of the node are labelled from left to right by the symbols that are used for replacing it. In the parse tree shown at Fig. 3.3(e), we can see that there are two interior nodes labelled as c_expression, and one interior node labelled as c_statement.
- The leaves of the parse tree are labelled by terminals. When the leaves are read from left to right, they constitute the ***yield*** of the tree. For example, in the parse tree shown in Fig. 3.3(e), the yield is count = count + 10;
- The parse tree is independent of the order of application of productions. For example, in the above derivation for *count = count + 10;*, the productions used were 1, 4, 3, and 2 in that order. The parse tree would be the same, even if we had used production 2 to replace the c_expression in the extreme right and then consequently used production 3 to replace the c_expression in the left.
- A pre-order traversal of the interior nodes of a parse tree brings out a unique leftmost derivation. The exact reverse of a post-order traversal of the interior nodes in a parse tree brings out a unique rightmost derivation. To recall, a post-order traversal of a tree is visiting the left sub-tree followed by right sub-tree followed by the root. For example, by observing the parse tree in Fig. 3.3(e) we can see that the unique leftmost derivation (pre-order) is as follows:

c_statement	=>	IDENTIFIER	EQ_TO_OP	c_expression			
	=>	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR	c_expression	SEMI_COLON
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	c_expression	SEMI_COLON
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	CONSTANT	SEMI_COLON
	=>	count	=	count	+	10	;

Similarly, we can deduce the unique rightmost derivation from the parse tree (reverse of post-order) in Fig. 3.3(e), as follows:

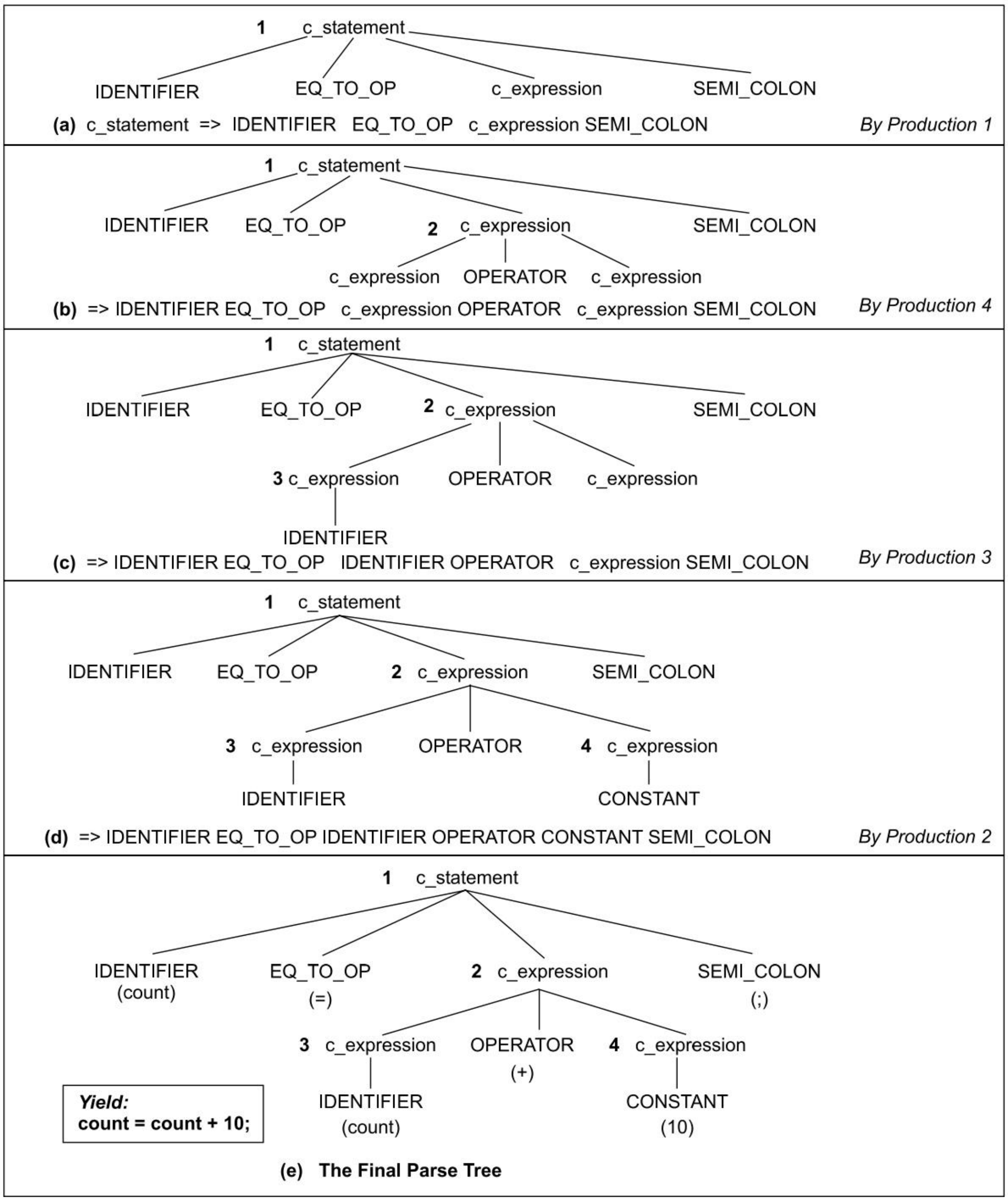

1 c_statement
IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
(a) c_statement => IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
By Production 1
1 c_statement
IDENTIFIER EQ_TO_OP 2 c_expression SEMI_COLON
c_expression OPERATOR c_expression
(b) => IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression SEMI_COLON
By Production 4
1 c_statement
IDENTIFIER EQ_TO_OP 2 c_expression SEMI_COLON
3 c_expression OPERATOR c_expression
IDENTIFIER
(c) => IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR c_expression SEMI_COLON
By Production 3
1 c_statement
IDENTIFIER EQ_TO_OP 2 c_expression SEMI_COLON
3 c_expression OPERATOR 4 c_expression
IDENTIFIER CONSTANT
(d) => IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR CONSTANT SEMI_COLON
By Production 2
1 c_statement
IDENTIFIER (count) EQ_TO_OP (=) 2 c_expression SEMI_COLON (;)
3 c_expression OPERATOR (+) 4 c_expression
IDENTIFIER (count) CONSTANT (10)
Yield: count = count + 10;
(e) The Final Parse Tree

Fig. 3.3 *Parse tree construction for count=count + 10;*

```
c_statement => IDENTIFIER  EQ_TO_OP   c_expression
            => IDENTIFIER  EQ_TO_OP   c_expression  OPERATOR  c_expression  SEMI_COLON
            => IDENTIFIER  EQ_TO_OP   c_expression  OPERATOR  CONSTANT      SEMI_COLON
            => IDENTIFIER  EQ_TO_OP   IDENTIFIER    OPERATOR  CONSTANT      SEMI_COLON
            =>    count        =         count         +         10             ;
```

Before we end the discussion on parse tree, let's see the advantages of representing a derivation by a parse tree:

- It makes the hierarchical syntactic structure of the sentence explicit. By viewing the final parse tree in Fig. 3.3(e), it is easy to make out the hierarchical structure of *count=count+10.*
- It illustrates the replacement of the productions clearly. Consider the C statement, *count = count + 10 / 5;,* as you can see later in the section, it can be derived in two ways. The parse trees shown in Fig. 3.4 can clearly tell us what replacements were made in either case.

A grammar is said to be ***ambiguous*** if it produces more than one parse tree for the same sentence. Ambiguous grammars produce more than one leftmost derivation or more than one rightmost derivation.

Consider a C statement '*count = count + 10 / 5;*'. Let's take a look at the leftmost derivation of this statement for checking its conformance with the grammar specified in Table 3.1.

```
c_statement => IDENTIFIER EQ_TO_OP c_expression                                              SEMI_COLON By P1
            => IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression                        SEMI_COLON By P4
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR c_expression                        SEMI_COLON By P3
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR c_expression OPERATOR c_expression SEMI_COLON By P4
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR CONSTANT     OPERATOR c_expression SEMI_COLON By P2
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR CONSTANT     OPERATOR CONSTANT     SEMI_COLON By P2
            =>    count       =       count        +        10           /         5            ;
```

Another way of doing leftmost derivation, where the order of productions used varies from the previous case.

```
c_statement => IDENTIFIER EQ_TO_OP c_expression                                              SEMI_COLON By P1
            => IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression                        SEMI_COLON By P4
            => IDENTIFIER EQ_TO_OP c_expression OPERATOR c_expression OPERATOR c_expression SEMI_COLON By P4
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR c_expression OPERATOR c_expression SEMI_COLON By P3
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR CONSTANT     OPERATOR c_expression SEMI_COLON By P2
            => IDENTIFIER EQ_TO_OP IDENTIFIER   OPERATOR CONSTANT     OPERATOR CONSTANT     SEMI_COLON By P2
            =>    count       =       count        +        10           /         5            ;
```

Figure 3.4 shows the parse trees for both of the above derivations.

Observe that there exists more than one parse tree with differing yields. This would create a problem at the time of generating a code. Hence, it is imperative that we disambiguate the grammar.

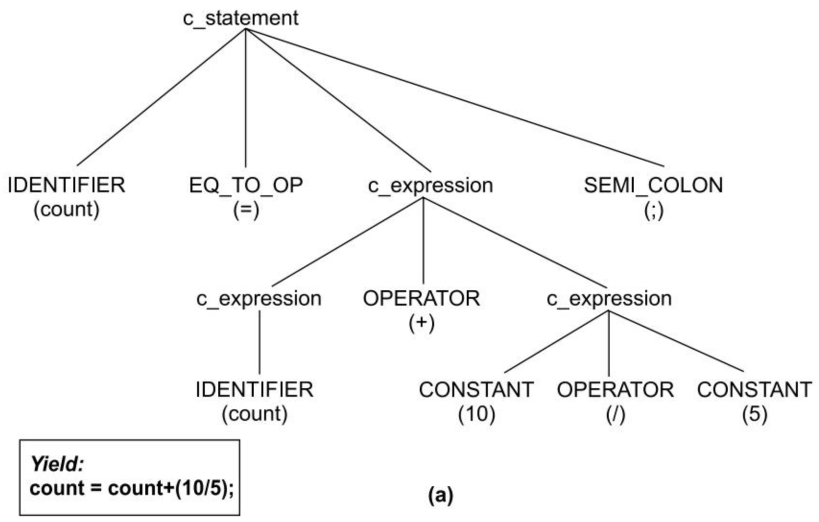
How do we disambiguate these grammars?

The fundamental idea behind the disambiguation of grammars is to eliminate some of the undesirable parse trees by means of specifying one or more disambiguating rules.

In order to eliminate ambiguity, operators like *,/,+,– are grouped hierarchically according to their ***precedence***. Precedence specifies the order of evaluation in an expression. Operations with higher precedence are carried out before operations having lower precedence. Table 3.3 shows a sample operator precedence in C language.

Table 3.3 *Operator precedence*

Operators	Priority
+ –	Lower priority
* /	Higher priority



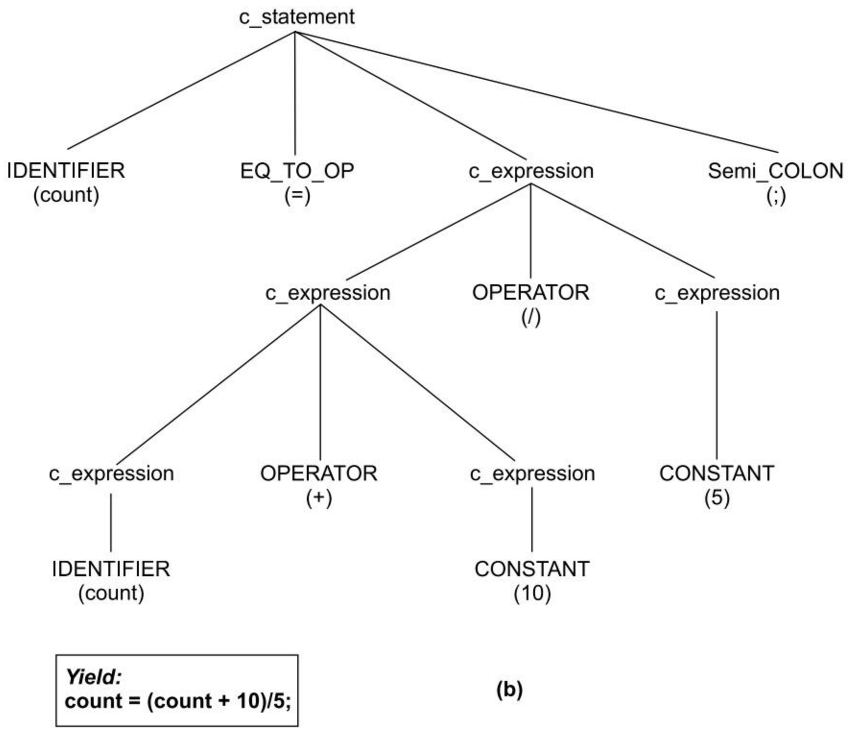


Fig. 3.4 *Parse trees for count=count + 10 /5 ;*

The operator precedence table shown in Table 3.3 will disambiguate statements like

```
count = count + 10/5;
```

This will be evaluated as

```
count = count + (10/5);
```

since division operator '/' is of higher precedence compared to '+' . It has eliminated the option of *count = (count + 10)/5;* since it gives an impression that plus operator has higher precedence than division operator.

The operator precedence table still does not disambiguate the cases like

```
var1 = var2 / var3 * 5;
```

Since both '*' and '/' have the same precedence. This can be evaluated as

```
var1 = (var2 / var3) * 5;
```

or

```
var1 = var2 / (var3 * 5);
```

This ambiguity is eliminated by specifying the order in which consecutive operations within same precedence group are carried on. This is called as ***associativity***. The operators of same precedence group can be evaluated from left to right (called as left to right associativity) or from right to left (called as right to left associativity). Using C language norm that each of the precedence group in Table 3.3 have left to right associativity then

```
var1 = var2 / var3 * 5;
```

would be evaluated to

```
var1 = (var2 / var3) * 5;
```

since the expression is evaluated from left to right for the same precedence group of '*' and '/' . This has eliminated the case of *var1 = var2 / (var3*5),* since it gives the impression that evaluation has happened from right to left. In any case, natural order of evaluation can always be altered by use of parenthesis.

The associativity and precedence rules mentioned above can also be reflected in the grammar itself, even though it is cumbersome and non-intuitive.

Before we end this section on context-free grammars, let's see the advantages of specifying the syntax of a language using context-free grammar.

- An efficient parser can be constructed automatically from a properly designed grammar. This is very similar to building a lexical analyser given the lexical specifications in regular expression as discussed in Chapter 2. There are tools like ***yacc***, ***bison*** that can convert syntax specification in CFG form to a parser. Section 3.1.2 shows an example of generating a parser from a CFG using the above-mentioned tools.
- Specialised tools like 'yacc' and 'bison' can perform verification and validation of CFG. They can detect flaws like ambiguities, unused rules, etc. in a grammar. This helps in correcting the grammar in early design phase of a language before we decide what action should follow on detection of a particular sequence.
- The context-free grammar lends structure to a program easing out translation to intermediate code, as we would see in the later chapters.

3.1.1 Using Context-free Grammar to Automatically Generate a Parser

The automatic generation of a parser from the CFG is perhaps the most important use of the context-free grammar. The context-free grammar is included as a part of the grammar-specifications file supplied to a parser generator utility like 'bison' in order to generate a parser. In this section, we discuss the structure of the grammar-specifications file and how the context-free grammar fits into it.

Listing 3.1 shows the grammar-specification file corresponding to the context-free grammar for simple C assignment statement described in Table 3.1.

```
%token IDENTIFIER CONSTANT OPERATOR EQ_TO_OP SEMI_COLON
%start c_statement
%{
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>

    extern int yylex();
    int yyerror(char *s);

%}

%%

c_statement   : IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
              ;

c_expression : CONSTANT
              ;

c_expression : IDENTIFIER
              ;

c_expression : c_expression OPERATOR c_expression
              ;

%%

extern int column;
int yydebug = 0;
char input_str[500];

int
yyerror (char *s)
{
    printf ("%s", input_str);
    fflush (stdout);
    printf ("\n%*s\n%*s\n", column, "^", column, s);
    return (1);
}

int
main (int argc, char **argv)
{
    int ret;

47        if (argc != 2) {
48            printf ("Usage: %s 'C statement' \n", argv[0]);
49            return (1);
50        }
51
52        strcpy (input_str, argv[1]);
53
54        ret = yyparse ();
55
56        if (ret == 0) {
57            printf ("%s", input_str);
58            printf ("\nSYNTAX CORRECT \n");
59        } else {
60            printf ("SYNTAX INCORRECT \n");
61        }
62
63        return (0);
64    }
```

Listing 3.1 *c-stmt-gram.y*

A grammar-specification file like the one illustrated in Listing 3.1 can be broadly divided into 3 parts.

```
Declarations
%%
Production Rules
%%
Auxiliary Functions
```

The declarations section consists of declarations of all the non-terminals (tokens) used in the grammar. This is illustrated in line 1 of Listing 3.1. The declarations section also contains the declaration of the start symbol that we discussed in Section 3.2. This is illustrated in line 2 of Listing 3.1, where we declare that the start symbol is c_statement. The declarations section can also contain a literal block of C code enclosed in {% and %} lines, exactly the way it is in the lexical specification file. This is illustrated from line 3 to 11 of Listing 3.1.

The production rules section consists of a list of grammar rules each separated by a semicolon (;). A colon (:) separates the left-hand and the right-hand sides of the productions. In the rules section, the first rule (line 15) defines the c statement. This is the production 1 of Table 3.1. The rules for c expression are mentioned next. These are the productions 2, 3, 4 of Table 3.1.

The auxiliary functions section consists C code that is copied verbatim into the generated code for parser. In the auxiliary section, we typically define *yyerror( )* function that is responsible for printing where the syntax error is found in case of erroneous input. This is shown from lines 33 to 40 in Listing 3.1. The auxiliary functions section also defines the `main()`, which in turn invokes the parsing routine *yyparse( )* at line 54. The return value of *yyparse( )* determines whether the given input is syntactically correct or otherwise. This is illustrated by line 56 in Listing 3.1.

3.1.2 Example 1—A Simple C Assignment Statement Checker Program

We now create a simple C assignment statement checker program that can verify the syntax of C statement using the grammar-specification file shown in Listing 3.1. The C assignment statement checker program takes the input C statement as a command line parameter and verifies if it is in conformance to the grammar specified in Listing 3.1.

The main components of C assignment statement checker program are:

1. A grammar-specification file (see Listing 3.1) based on the context-free grammar for C assignment statement specified in Table 3.1.
2. A lexical specifications file (not listed here) that describes a lexical analyser returning the tokens IDENTIFIER, CONSTANT, SEMI_COLON, OPERATOR and EQ_TO_OP.
3. A *main ()* routine that invokes the parsing routine – *yyparse( )*. This is part of the auxiliary functions section of the grammar-specification file given in Listing 3.1.

The following dialog shows how we build the c statement parser and execute it. We can see from the dialog that the parser generator utility ***'bison'*** takes in the grammar specifications file *c-stmt-gram.y* and generates the parser. The lexical analyser is also built automatically from the lexical specifications file.

```
# Generating the Parser from Grammar Specifications
$ bison -dy -oc-stmt-gram.c -v c-stmt-gram.y
conflicts: 1 shift/reduce

# Compiling the Parser
$ gcc -g -Wall -DGENERATED_PARSER -c -o c-stmt-gram.o c-stmt-gram.c

# Generating the Lexical Analyser from Lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the Lexical Analyser
$ gcc -c -DGENERATED_PARSER -DCHAP3_EX1 -o c-stmt-lex1.o c-stmt-lex.c

# Building ex1 Binary
$ gcc c-stmt-gram.o c-stmt-lex1.o -o ex1

# Variant 1
$ ./ex1 'count=5;'
count=5;
SYNTAX CORRECT

# Variant 2
$ ./ex1 'count=index;'
count=index;
SYNTAX CORRECT

# Variant 3
$ ./ex1 'count=3*2;'
count=3*2;
SYNTAX CORRECT


# Variant 4
```

```
$ ./ex1 'count=count+index;'
count=count+index;
SYNTAX CORRECT

# Variant 5
$ ./ex1 'count=count+1;'
count=count+1;
SYNTAX CORRECT

# Variant 6
$ ./ex1 'count=count*2+5;'
count=count*2+5;
SYNTAX CORRECT

# Variant 7
$ ./ex1 'count=count*2+index;'
count=count*2+index;
SYNTAX CORRECT

# Missing Identifier / Constant
$ ./ex1 'count=5+;'
count=5+;
        ^
syntax error
SYNTAX INCORRECT

# Missing semicolon
$ ./ex1 'count=index'
count=index
          ^
syntax error
SYNTAX INCORRECT
```

The reader is advised to try out some more invocations of *ex1* with various combinations of identifiers and constants to gain better understanding of the grammar.

3.2 CLASSIFICATION OF PARSING TECHNIQUES

In the last section we learnt about what is context-free grammar and how it helps in defining the syntax of the language. This section presents an overview of various techniques used to parse the input in accordance to a defined grammar. The consequent sections describe in detail the parsing techniques mentioned here.

Parsing techniques can be classified into two major categories ***top-down parsing*** or ***bottom-up parsing*** (see Fig. 3.4) depending on how the parse tree is built. The top-down parsers build the parse tree starting from the root node and work down to the leaves. The bottom-up parsers build the parse tree starting from the leaves and goes all the way to the root. Top-down parsing is described in Section 3.4, while bottom-up parsing is the focus of Section 3.5.

From applicability point of view, the largest class of grammars for which top-down parsers can work successfully is called as ***LL grammars*** (scanning **L**eft to right, **L**eftmost derivation). The largest class of grammars for which bottom-up parsers can succeed is called ***LR grammars*** (scanning **L**eft to right, **R**ight most derivation).

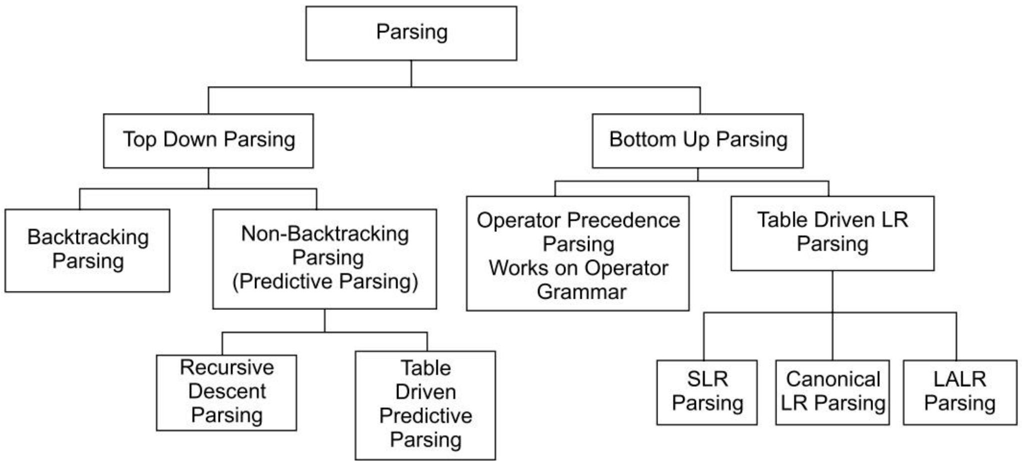

Parsing
Top Down Parsing
Bottom Up Parsing
Backtracking Parsing
Non-Backtracking Parsing (Predictive Parsing)
Operator Precedence Parsing Works on Operator Grammar
Table Driven LR Parsing
Recursive Descent Parsing
Table Driven Predictive Parsing
SLR Parsing
Canonical LR Parsing
LALR Parsing

Fig. 3.5 *Classification of parsing techniques*

Backtracking parsing is one of the top-down techniques where, if we make a sequence of erroneous expansions and subsequently discover a mismatch, we undo the effects and roll back the input pointer. A backtracking top-down parsing example is discussed in Section 3.4. The top-down backtracking parsers are seldom used in practice owing to several disadvantages it entails (see Section 3.4).

One way of compensating the need for backtracking is to use the next input symbol to 'guide' the parser to use the correct production rule. Thus knowing the current non-terminal to be expanded and the next input symbol, the parser makes an informed decision as to what production needs to be expanded. This parsing technique is called ***predictive parsing.*** There are certain transformations that grammar needs to undergo, in order to make it suitable for predictive parsing. This is discussed in detail in Section 3.4. There are two ways of implementing a top-down predictive parser. One of the methods involves having a procedure for each non-terminal. These procedures are responsible for parsing the constructs defined by its non-terminal. This method is called ***recursive descent parsing*** owing to the recursive nature of the resultant procedures. This is detailed in Section 3.4.1. The other method of implementing a top-down predictive parser is to have a stack maintained explicitly by the parser. The parser determines the next production to be applied by using a parsing table. The parsing table is a table of production rules, which can be indexed using the current non-terminal being expanded and the next input symbol. The parsing table is populated using the grammar of the language. This method is known as ***table-driven predictive parsing*** owing to the use of parsing table as a mechanism for determining the next production to be used. This forms the focus of discussion in Section 3.4.2.

In bottom-up parsing the parse tree is built from the leaves to the top. In bottom-up parsing, the input is 'reduced' to the start symbol in a sequence of steps known as the reduction steps. Each reduction step involves matching of a particular string on the right-hand side of the production and replacing it by the left-hand side of the production.

Bottom-up parsers are usually implemented using a stack. The next input symbol is examined to make a decision to:

(a) ***shift*** the next input symbol on to the top of stack or

(b) pop the top few elements and push a non-terminal equivalent of the same on to the top of stack (called as ***reduce operation***).

Owing to this shift or reduce operation, bottom-up parsing is also called as ***shift reduce parsing***.

The choice of whether to shift the next input symbol or reduce the processed input is based on a table called as precedence relations table for ***operator precedence parsing***.

The LR parsers, i.e. SLR, canonical LR and LALR use a ***parsing table*** derived out of the grammar for deciding whether to shift the next input symbol or reduce the processed input. The method used for deriving these tables from grammar is different for each one of them, viz. SLR, canonical LR and LALR. The SLR (***simple LR***) method succeeds in generating a parsing table out of the grammar for a sub-set of LR grammars called SLR grammars. The ***canonical LR parsing*** method is the most powerful method of generating the parsing table from the given grammar. It succeeds in generation of parsing table for a largest sub-set of LR grammars called as LR(1) grammars. The disadvantage of the canonical LR parsing method of generating the parsing table from the grammar is that the size of the parsing table is very large. LALR (***look ahead LR***) method tries to shrink the parsing table derived out of canonical LR parsing method by applying some considerations. In the process of doing so, it loses some of the power of the canonical LR method. However, it succeeds to a pretty reasonable sub-set of LR grammars. The LALR method of shift reduce parsing is used in most of the popular parser generators like ‘bison’ and ‘yacc’.

3.3 ERROR REPORTING AND RECOVERY IN SYNTAX ANALYSER

One of the main functions of the syntax analyser is to report errors in the input source program and emit an informative diagnostic message. The diagnostic message should typically give out line number of the offending line in the source program and also emit information related to the error (e.g. in a C source program, it could be a missing semicolon). This is called ***error reporting*** in syntax analyser. The syntax analyser should not stop at the first instance of encountering an error, it should continue and report as many errors present in the input source program as possible, so that the programmer can correct all of the errors in the input source program. This function of recovering from the first error and continuing to process the rest of the input source program is known as ***error recovery*** in a syntax analyser. The error reporting and recovery is performed by a logical entity in the syntax analyser called as ***error handler***.

The error handler module of a syntax analyser has two main functions: (1) It should report the errors in the input source program with enough information to enable the programmer to correct the mistake. (2) The error handler should not stop at the first error encountered in the input source program. It should recover and report as many errors present in the input source program as possible. Incorporating the intelligence in the error handler to accomplish the above functions should not slow down the syntax analysis phase significantly.

Let us take an example C program and use the GNU’s C compiler ‘gcc’ to demonstrate the error reporting and error recovery concepts.

```
# An input C source Program containing errors
$ cat -n test1.c
     1  #include <stdio.h>
     2
     3  int main()
     4  {
     5          int var1,var2;
     6
```

```
 7
 8          var1 = 0;
 9          var2 = 10;
10
11          printf("This is message 1 ")
12
13          var1 = var2 ;
14
15          for( i = var1; i < var2; i++){
16               printf("This is iteration %d ",i);
17          }
18   }
```

The input C source program *test1.c* has two errors. (1) There is a missing semicolon in line 11 and (2) the variable 'i' used in line 15 has not been declared earlier.

The dialog below shows how the GNU's C compiler 'gcc' parses the above program.

```
$ gcc test1.c -o test1
test1.c: In function 'main':
test1.c:13: parse error before 'var1'
test1.c:15: 'i' undeclared (first use in this function)
test1.c:15: (Each undeclared identifier is reported only once
test1.c:15: for each function it appears in.)
```

The parser in gcc has reported the error in line 13 before the variable 'var1', which is nothing but the end of line 11. This is indicative of missing semicolon in line 11. Note that the parser of *gcc* did not stop there, it continued parsing the subsequent lines of input source program and identified an error in line number 15. The parser in *gcc* has performed error recovery from earlier error in line 13 and continued parsing. The error reporting on line number 15 clearly says that 'i' is not declared. Note that, the parser was smart enough to report the non-declaration of 'i' once, despite being used more than once.

The above example demonstrates the error reporting and error recovery features of a parser.

The main considerations in error reporting are:

- The error handler should report the place in the input source program, where the error has occurred. The offending line number should be emitted for the programmer to correct the mistake.
- The diagnostic message emitted out by the error handler module of the parser should give out enough information to help the programmer correct the mistake in the input source program.

The job of error recovery for the error handler is trickier. The following are some of the considerations in error recovery:

- The error recovery should not be partial where spurious errors not made by the programmer are falsely identified as errors and displayed.
- The error recovery should also be cautious not to get into a bind when a totally unexpected input is given.
- The compiler designer needs to decide if error repair feature should be incorporated in the error handler. Usually error repair is not very cost-effective except in situations where the input source program is from beginners to programming.

There are several error-recovery strategies that can normally be applied in the error handler of a parser. They are:

1. Panic mode recovery
2. Phrase level recovery
3. Error productions
4. Global corrections

Panic mode recovery is the simplest of all the strategies mentioned above. On discovering an error, the parser discards all the input symbols until it encounters one of the symbols in a designated set of ***synchronisation tokens***. The symbols in synchronisation set are determined by the characteristics of the language. For example, in C language, a semicolon, which indicates the end of a C statement, is a good candidate for being an element in synchronisation token set. The challenge for the compiler designer is to select the right elements to form the synchronisation set. The advantage of panic mode recovery is that it is simple and easy to implement. The disadvantage in panic mode recovery is that the parser could potentially end up ignoring a considerable amount of input without looking for errors. This method of error recovery is ideally suited for situations where multiple errors in the same line are rare.

In phrase level error recovery, the parser on encountering an error attempts to perform a local correction in the rest of the input, which allows it to continue. For example, the parser could add a missing semicolon or convert an existing comma to semicolon, etc. so that the parser can continue parsing further input. The phrase-level error recovery could lead the parser into an infinite loop especially if a wrong choice of correction is attempted before the current input symbol. This technique is used commonly in error-repairing compilers. This method of error recovery is suited in situations where the error is detected as soon as it happens. It cannot work well in situations where there is a certain lag between error detection and error occurrence.

In error productions method of error recovery, there are additional productions defined in the language grammar for catching errors. The action for those erroneous productions would be to flash an informative error message relevant for the error production. Defining the error productions without overlapping the legal productions of the language presents a challenge to the compiler designer.

In global correction error recovery technique the partial parse tree of the erroneous input string is compared to the parse tree of a related correct input strings and the distance (i.e. the additions, deletions, etc.) between them is computed. The parse tree with minimum distance from the erroneous input string is picked as the correct input string and the corrections are then made appropriately to the input string. The distance between incorrect input string and the correct ones are computed by certain specialised algorithms. This method of error recovery is very expensive in term of memory and time and hence it remains of theoretical interest.

We shall see the suitability of an error recovery scheme to a parsing technique, during the detailed discussion of various parsing techniques in the succeeding sections.

3.4 TOP-DOWN PARSING

In this section we study in detail about top-down parsing. Initially we shall discuss about a generic form of top-down parsing. Next, we address the kinds of grammars that are suitable for top-down parsing followed by a brief discussion on how to transform grammars to suit top-down parsing. After that we study two techniques of implementing top-down parser namely the recursive descent and table-driven predictive parsing.

Top-down parsing is an attempt to find the leftmost derivation for an input string. It attempts to construct a parse tree for the input starting from the root (start symbol) and creating the nodes of the parse tree in pre-order. In order to make the concept clear, let us consider a C statement

```
count = index + 30;
```

and examine how it is derived by a top-down parser using the production rules given in Table 3.2.

c_statement	=>	IDENTIFIER	EQ_TO_OP	c_expression			SEMI_COLON	*By P1*
	=>	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR	c_expression	SEMI_COLON	*By P4*
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	c_expression	SEMI_COLON	*By P3*
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	CONSTANT	SEMI_COLON	*By P2*
	=>	count	=	index	+	10	;	

The parse tree for the above derivation is shown in Fig. 3.6.

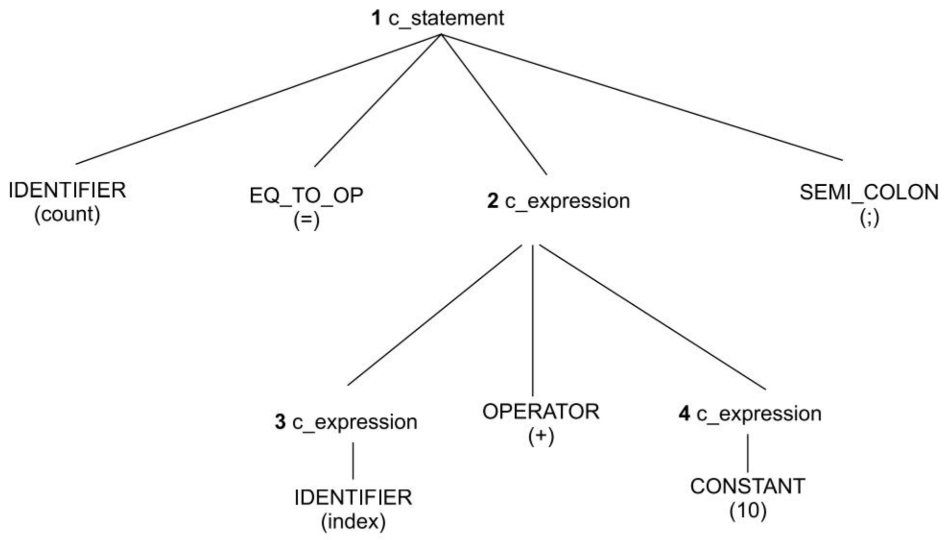


Fig. 3.6 *Top-down parsing for count=index + 10;*

In ***backtracking top-down parsers***, we begin with the start symbol and apply productions until we get the desired string. The choice of which production needs to be applied depends on the order of production rules specified in the grammar. If this choice leads to a dead-end, the parser would have to backtrack to that decision point moving backwards through the input and start again making a different choice. This goes on until the parser ends up with the appropriate production for matching the complete input or it runs out of choices. The following example will illustrate the concept.

Consider the grammar for a single C statement presented in Table 3.1, reproduced below for convenience.

c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression	→	CONSTANT
	|	IDENTIFIER
	|	c_expression OPERATOR c_expression

The table below shows how an input string *'count = index + 100;'* can be parsed using the top-down parser that can do backtracking. Column (A) shows the expansion thus far, column (B) shows the remaining input and column (C) shows the action attempted at each step.

(A) Expansion till now	(B) Remaining input	(C) Action attempted
	count = index + 100 ;	Try expanding a production with the start symbol - c_statement as LHS. There is only 1 production with the start symbol on the LHS, i.e. Production 1. We start with that c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
IDENTIFIER EQ_TO_OP c_expression SEMI_COLON	count = index + 100 ;	Match count with IDENTIFIER. Successful.
EQ_TO_OP c_expression SEMI_COLON	= index + 100 ;	Match = with EQ_TO_OP. Successful.
c_expression SEMI_COLON	index + 100 ;	Try expanding production 2, c_expression → CONSTANT
CONSTANT SEMI_COLON	index + 100 ;	index does not match CONSTANT. Dead-end, backtrack
c_expression SEMI_COLON	index + 100 ;	Try expanding production 3, c_expression → IDENTIFIER
IDENTIFIER SEMI_COLON	index + 100 ;	Match index with IDENTIFIER. Successful.
SEMI_COLON	+ 100 ;	SEMI_COLON does not match + . Dead-end, backtrack
c_expression SEMI_COLON	index + 100 ;	Try expanding Production 4, c_expression →c_expression OPERATOR c_expression
c_expression OPERATOR c_expression SEMI_COLON	index + 100 ;	Try expanding production 2, c_expression → CONSTANT
CONSTANT OPERATOR c_expression SEMI_COLON	index + 100 ;	index does not match CONSTANT. Dead-end, Backtrack
c_expression OPERATOR c_expression SEMI_COLON	index + 100 ;	Try expanding production 3, c_expression → IDENTIFIER
IDENTIFIER OPERATOR c_expression SEMI_COLON	index + 100 ;	Match index with IDENTIFIER. Successful
OPERATOR c_expression SEMI_COLON	+ 100 ;	Match + with OPERATOR. Successful.
c_expression SEMI_COLON	100 ;	Try expanding production 2, c_expression → CONSTANT
CONSTANT SEMI_COLON	100 ;	Match 100 with CONSTANT. Successful.
SEMI_COLON	;	Match ; with SEMI_COLON . Successful.
		Success

From the steps above, we can see that whenever we reach a dead-end, we backtrack to the last decision point undo the decision and try another production. If all the productions are exhausted, we backup to the preceding decision point and so on. This procedure continues until the complete input is matched or we have exhausted all combinations.

The top-down backtracking parsing procedure that we just described can cope up with any kind of grammar. However, when there are a large number of non-terminals, the algorithm becomes very slow due to the combinatorial explosion in terms of the productions that have to be tried out before arriving at the correct parse.

If we make a series of expansions and subsequently discover a mismatch, we may have to undo the semantic effects of erroneous expansions like, say the removal of entries from symbol table. Undoing the semantic effects is a substantial overhead for a compiler. This is one of the major disadvantages of backtracking a top-down parser.

In top-down backtracking parser, the order in which alternates of production rules are tried can also affect the language considered. This makes backtracking parsers unsuitable for production compilers.

Another disadvantage of the backtracking top-down parser is the difficulty in error reporting. In top-down backtracking parser, it is difficult to pinpoint where the error has occurred and consequently the compiler cannot emit informative error messages.

Backtracking can be avoided by transforming the grammar in such a way that at each step the choice of production that can lead us to solution can be easily identified. In other words, at each step, we can 'predict' which of the productions can lead us to the complete derivation of the input string, if one exists. The idea behind a top-down ***predictive parser*** is that the current non-terminal being processed combined with the next input symbol can guide the parser to take the correct production rule eventually leading to the match of complete input string. The predictive parser is a type of top-down parser that does not require backtracking in order to derive various input strings. This is possible because the grammar for the language is transformed such that backtracking is not needed.

What kind of transformations do we make to the grammar rules to suit a predictive parser? There are two types of transformations done to the grammar in order to suit a predictive parser. They are:

1. elimination of left recursion
2. left factoring

Elimination of left recursion is a grammar transformation that is used for producing grammar suitable for predictive parsing. Let's take an example to get an idea of why the transformation is required and how we do the transformation.

Consider an erroneous input string (real programmers do type mistakes!) for a parser using the production rules in Table 3.1

```
count = -;
```

Let's see how the parsing proceeds using the production rules in Table 3.1.

The top-down parser will first match the production rule 1 and consume the input `count` and `EQ_TO_OP`. On receiving '-', it tries to verify whether it is a C expression. How does the parser do it? It checks if the input (which is the – i.e. OPERATOR) fits production 2 first. It does not. Next, we try production 3, it does not match either, since production 3 starts with IDENTIFIER. Next, we try production 4, the parser will try to match if '-' is a c_expression, which is what we started in the first place. This leads to us try and match '-' recursively for c expression, without really consuming input.

How do we solve this problem?

Observe from production 2, 3 and 4 that the c_expression has to begin with IDENTIFIER or CONSTANT (because that is where the recursion has to end). The solution to the problem is to take advantage of the fact that the c_expression has to definitely start with either a terminal (CONSTANT or IDENTIFIER in this case). We can rewrite the left recursive production 4 to reflect this philosophy.

c_expression* → *IDENTIFIER c_expr_rest	……….. (3.5)
c_expression* → *CONSTANT c_expr_rest	……….. (3.6)

c_expr_rest* → *OPERATOR IDENTIFIER c_expr_rest	……….. (3.7)
c_expr_rest* → *OPERATOR CONSTANT c_expr_rest	……….. (3.8)
c_expr_rest → ∈ */* empty string */*	……….. (3.9)

Note that (3.5) in conjunction with (3.9) is exactly equivalent to production 2 (c_expression → IDENTIFIER) and (3.6) in conjunction with (3.9) is equivalent to production 3 (c_expression → CONSTANT). The transformed grammar is reflected in Table 3.4.

Let's see how the top-down parsing of the input happens with the new transformed rules. The top-down parser will first match the production rule 1 and consume the input *count and EQ_TO_OP.* On receiving '-', it tries to verify whether it is a C expression. It checks if the input (which is the – operator) fits production 2 or 3 in that order. It does not match since '-' is not a CONSTANT or IDENTIFIER as required by these rules. The parser then decides that it is not conformant with the grammar specified in Table 3.4 and rejects the input string.

Table 3.4 *Productions after elimination of left recursion*

1	c_statement	→	IDENTIFIER	EQ_TO_OP	c_expression	SEMI_COLON
2	c_expression	→	CONSTANT	c_expr_rest		
3		|	IDENTIFIER	c_expr_rest		
4	c_expr_rest	→	OPERATOR	IDENTIFIER	c_expr_rest	
5		|	OPERATOR	CONSTANT	c_expr_rest	
6		|	∈			

Left factoring is another transformation that is commonly applied to the grammar to make it suitable for predictive parsing. We shall look at an example to get an idea of why the transformation is required and then we shall see how the transformation is done.

Consider an input string *'result = total + 20;'* for conformance with the grammar in Table 3.4. After we consume *result* and *EQ_TO_OP,* we would get the input of *total*, which is an IDENTIFIER. The IDENTIFIER would match the production 3, and would be consumed by the parser. The next input symbol would be the '+' operator. The parser would try and match it with the c_expr_rest. At this point, the parser would be in dilemma whether to use production 4 or production 5 to expand. Both of them start with an OPERATOR. By hard coding the parser to choose one of the productions all the time, we could potentially be reporting legal input strings not to be conformant to the grammar.

How can we 'predict' which production rule to take for complete derivation of input string?

A common approach to the problem is to rewrite the grammar to factor the common prefix (which is an OPERATOR in this case) out of the two alternatives (production 4 and 5 in this case) and create a new rule.

If we left-factor the rules 4 and 5 we get

c_expression → OPERATOR c_expr_factor	……….. (3.10)

c_expr_factor → IDENTIFIER c_expr_rest	……….. (3.11)
c_expr_factor → CONSTANT c_expr_rest	……….. (3.12)

The new transformed rules after the left factoring is given in Table 3.5.

Table 3.5 *Productions after (a) elimination of left recursion and (b) left factoring*

1	c_statement	→		IDENTIFIER	EQ_TO_OP	c_expression	SEMI_COLON
2	c_expression	→		CONSTANT	c_expr_rest		
3			|	IDENTIFIER	c_expr_rest		
4	c_expr_rest	→		OPERATOR	c_expr_factor		
5			|	∈			
6	c_expr_factor	→		IDENTIFIER	c_expr_rest		
7			|	CONSTANT	c_expr_rest		

Let's see how the parsing proceeds with the new transformed rules. After, the top-down parser consumes *result* and *EQ_TO_OP,* the next input is *total*, which is an IDENTIFIER. The IDENTIFIER would match the production 3, and would be consumed by the parser. The next input symbol would be the '+' OPERATOR. The parser would try and match it with the c_expr_rest. The top-down parser would make an attempt to match production 4 and consume the OPERATOR. The next input symbol is 20. The parser would try and match with one of the productions for c_expr_factor, which is 6 or 7. The input symbol '20' matches the CONSTANT, so production 7 is chosen. The next input symbol is ';' which will be tried against c_expr_rest. The production 5 (empty string) would match since the ';' is not a part of any of the rules for c_expr_rest. By doing this ***left factoring*** we effectively postponed the decision of choosing a production until a point is reached where it can be made properly (in this case after consuming the OPERATOR symbol).

Here is a generalisation of left factoring. consider a production

$$A \rightarrow \alpha\beta 1 \mid \alpha\beta 2$$

and an input that begins with a non-empty string derived from α, it is not clear whether to expand A to αβ1 or αβ2. We can solve this by deferring the decision by expanding A to αA^1. Then after seeing the input derived from α, we expand A^1 to β1 or β2. The left-factored form of original productions becomes

$$A \rightarrow \alpha A^1$$
$$A^1 \rightarrow \beta 1 \mid \beta 2$$

The grammars that are suitable for predictive parsing (after elimination of left recursion and Left factoring) are called as LL(k) grammars, where the first 'L' implies that we scan from left to right, the second 'L' stands for the leftmost derivation and 'k' stands for number of tokens of look ahead. In practice, LL(1) grammars are used for building programming languages.

We have understood how grammars are transformed to suit predictive parsers. Next, we see how to implement top-down predictive parsers.

There are 2 main techniques for implementing top-down predictive parsers. They are:

1. recursive descent parsing
2. table-driven predictive parser.

The next two sections describe these techniques in detail.

3.4.1 Recursive Descent Parsing

A recursive descent parser is a collection of procedures one for each non-terminal. Each procedure is responsible for parsing the constructs defined by its non-terminal. The syntax of most of the programming languages is recursive, the resulting procedures are also recursive, hence the name recursive descent parser. For the production rules mentioned in Table 3.5, there would be four functions c_statement, c_expression, c_expr_rest, and c_expr_factor denoting each of the non-terminals in the grammar. These functions are then constructed using the grammar in a straightforward way. For example, consider the c_statement.

The rule

c_statement	→	IDENTIFIER	EQ_TO_OP	c_expression	SEMI_COLON

could be translated into a function as shown below.

Table 3.6 *Pseudo-code for c_statement*

```
c_statement()
{
       if  ( match (IDENTIFIER)){
                 if(match (EQ_TO_OP)){
                        if(c_expression()){
                                if(match(SEMI_COLON)){
                                      return(SUCCESS);
                                }
                        }
                 }
       }
       return(FAILURE);
}
```

The function `match` checks if the current token is the same as the expected token.

Table 3.7 *Psuedo-code for 'match'*

```
match(expected)
{
      if(current_token == expected){
            current_token = get_next_token();
            return(SUCCESS);
      }
      return(FAILURE);
}
```

Similarly, for c_expression,

c_expression	→	CONSTANT	c_expr_rest
	|	IDENTIFIER	c_expr_rest

the pseudo-code is derived from the grammar as follows:

Table 3.8 *Pseudo-code for c_expression*

```
c_expression()
{
        if ( match (CONSTANT)){
                if ( c_expr_rest ()){
                        return(SUCCESS);
                }
        } else if (match ( IDENTIFIER)){
                if ( c_expr_rest ()){
                        return(SUCCESS);
                }
        }
        return(FAILURE);
}
```

The pseudo-code for c_expr_rest derived from the following grammar:

c_expr_rest	→	OPERATOR	c_expr_factor
	|	$\in$	

is shown below:

Table 3.9 *Psuedo-code for c_expr_rest*

```
c_expr_rest()
{
        if ( match (OPERATOR)){
                if ( c_expr_factor ()){
                        return(SUCCESS);
                }
        } else {
                return(SUCCESS); /* empty string */
        }
        return(FAILURE);
}
```

The pseudo-code for c_expr_factor derived from the following grammar:

c_expr_factor	→	IDENTIFIER	c_expr_rest
	|	CONSTANT	c_expr_rest

Table 3.10 *Pseudo-code for c_expr_factor*

```
c_expr_factor()
{
        if ( match (IDENTIFIER)){
                if ( c_expr_rest ()){
                        return(SUCCESS);
                }
        } else if ( match (CONSTANT)){
                if ( c_expr_rest ()){
                        return(SUCCESS);
                }
        }
        return(FAILURE);
}
```

Recursive descent parsers are simple and easy to implement. Most of the hand-written parsers are usually recursive descent parsers due to the intuitive nature of the technique. A recursive descent parser needs a large stack due to the recursion of the procedures. This could be prohibitive in systems where memory is at premium. Recursive descent parsers cannot be automatically generated easily from the grammar, compared to table-driven approaches, which we shall discuss later.

A sample recursive descent parser built for C statement syntax verification is presented in the next section to reinforce the concepts discussed till now.

3.4.1.1 *Example 2—A Recursive Descent Parser for Simple C Statement*

In this section, we build a recursive descent parser for verifying the syntax of c statements using the pseudo-code given earlier. The following dialog shows how we make the recursive descent parser for c statement and how we use it to verify syntax of c-statements.

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the Lexical Analyzer
$ gcc -c -o c-stmt-lex.o c-stmt-lex.c

# Building ex2 Binary
$ gcc -g -Wall recur_descent.c c-stmt-lex.o -o ex2

# Variant 1
$ ./ex2 'count=5;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=5;
SYNTAX CORRECT

# Variant 2
$ ./ex2 'count=index;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
```

```
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
count=index;
SYNTAX CORRECT

# Variant 3
$ ./ex2 'count=3*2;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=3*2;
SYNTAX CORRECT

# Variant 4
$ ./ex2 'count=count+index;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
count=count+index;
SYNTAX CORRECT

# Variant 5
$ ./ex2 'count=count+1;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=count+1;
SYNTAX CORRECT

# Variant 6
$ ./ex2 'count=count*2+5;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=count*2+5;
SYNTAX CORRECT

# Variant 7
$ ./ex2 'count=count*2+index;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
```

```
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
count=count*2+index;
SYNTAX CORRECT

# Missing Identifier / Constant
$ ./ex2 'count=5+;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
Error Error Error Error
SYNTAX INCORRECT

# Missing semicolon
$ ./ex2 'count=index'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
Error
SYNTAX INCORRECT
```

The following are some of the observations that can be made for a recursive descent parser.

- The recursive descent parser starts by calling the lexical analyser to get the first token. The procedure corresponding to the start symbol is invoked. The completion of this procedure indicates the parse is complete.
- A global variable holding the token that is not yet consumed by parser is used to predict which production needs to be expanded
- Printing a good error recovery message is not easy, since it is difficult to determine from the current routine, as to what is the calling routine and its context.

3.4.2 Table-driven Predictive Parsing

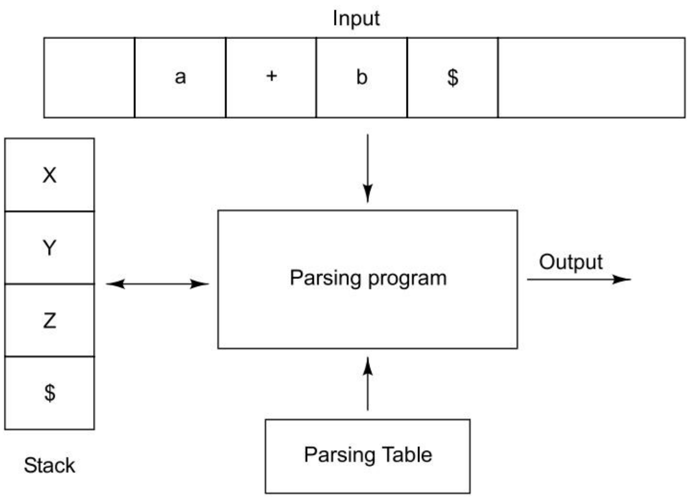
The recursive descent parser works by maintaining the production expansion implicitly on the stack by virtue of recursion in the procedures. The table-driven predictive parser maintains the stack explicitly. In recursive descent parser, the choice of which production needs to be applied is determined by the code structure. In the table-driven parser, the choice of which productions needs to be applied is determined by indexing into a parsing table, given the current non-terminal being expanded and the next input terminal.

Table-driven predictive parser consists of:

1. An input buffer that contains the string to be parsed followed by a $, a symbol used for indicating the end of input.
2. A stack containing the sequence of grammar symbols with a $ at the bottom of the stack.
3. A parsing table containing the production rules to be applied. This is a two-dimensional array M[A, a], where A is the non-terminal and 'a' is a terminal or the symbol $. This is a manifestation of the grammar rules.

4. A parsing program that takes the input string and determines if it is conformant to the grammar. The parser program uses the parsing table and the stack to arrive at this decision. The logic built into the parsing program does not change with the grammar.

The figure illustrates the various components of a table-driven parser.



The parsing program determines the action of the parser depending on

- X, the symbol on the top of the stack
- a, the current input symbol

There are 3 combinations of X and a, that are important for the parsing program. Each of those 3 conditions marked A, B and C are shown below. Any other condition other than these 3 denotes a parsing error.

#	Condition	Action
A	X = a = $	The parser announces the successful completion of parsing and returns.
B	X is a terminal and X = a ≠ $	The parser pops off the stack and advances the input pointer to the next input symbol.
C	X is a non-terminal	The program consults the entry M[X, a] in the parsing table. This entry is a production for X or an error entry. If M[X, a] is a production X → UVW, then the program replaces the X on the top of the stack by WVU (U on the top of stack). The parser can output the production or execute the code specified in the grammar. If M[X, a] is an error entry, then the program calls for an error recovery routine.

The parsing program discussed above is summarised in Algorithm 3.1.

```
tbl_driven_parse()
{
        a = get_next_tok();
        do
        {
                X = The top element of the Stack
                if( (X == $) or ( a == $ ))
                {
                        if ( X == a)
                                return(SUCCESS);
                        else
                                return(FAILURE)
                }

                if ( X is a terminal) {
                        if ( X is a){
                                pop X from the Stack and remove a from the input
                                a = get_next_tok()
                        } else {
                                return(FAILURE)
                        }
                } else {/* Non terminal */
                        if ( M[X][a] is a rule X -> Y1Y2…Yk){
                                pop X from the Stack
                                push Yk…Y2Y1 on the stack
                        } else {
                                return(FAILURE)
                        }
                }
        } while ( 1) /* Always */
}
```

Algorithm 3.1 *Table-driven predictive parsing*

The main data structure on which Algorithm 3.1 depends is the parsing table M. For a grammar described in Table 3.5, the parsing table M is shown in Table 3.11.

Table 3.11 *Parsing table for the grammar described in Table 3.5*

	IDENTIFIER	CONSTANT	OPERATOR	EQ_TO_OP	SEMI_COLON	$
c_statement	c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON					
c_expression	c_expression → IDENTIFIER c_expr_rest	c_expression → CONSTANT c_expr_rest				
c_expr_rest			c_expr_rest → OPERATOR c_expr_factor		c_expr_rest → ε	
c_expr_factor	c_expr_factor → IDENTIFIER c_expr_rest	c_expr_factor → CONSTANT c_expr_rest				

From the parsing table given in Table 3.11, we can understand the following:

- if c_statement has to be expanded and we encounter an input of IDENTIFIER, we use the production c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
- if c_expression has to be expanded and we encounter an input of IDENTIFIER, we use the production c_expression → IDENTIFIER c_expr_rest
- if c_expression has to be expanded and we encounter an input of CONSTANT, we use the production c_expression → CONSTANT c_expr_rest
- All the empty entries denote errors

Using the grammar of Table 3.5, and the corresponding parsing table given in Table 3.11, you can see how the predictive table-driven parser algorithm works for an input of

```
c = i + 5;
```

Initially in table-driven predictive parsing, the end of the input marker $ along with the start symbol (c_statement) is pushed on to the stack.

Stack **Remaining Input**

$	c_statement

c = i + 5 ; $

At this point the symbol on the top of the stack X is c_statement and the next input symbol is 'c' — an IDENTIFIER. Since the top of the stack is a non-terminal (condition C), we check for the entry in the parsing table at M[c_statement] [IDENTIFIER]. It contains the entry `c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON`. Hence we pop c_statement from the stack and push the RHS of the production in the reverse order. The order of pushing is such that IDENTIFIER would be on the top. Note that the input pointer is not advanced and hence it would continue to point to 'c', which is an IDENTIFIER. The figure below shows the stack expanding to the right with the top of the stack being its rightmost element.

Stack **Remaining Input**

$	SEMI_COLON	c_expression	EQ_TO_OP	IDENTIFIER

c = i + 5 ; $

At this point, the symbol on the top of the stack, X is an IDENTIFIER. The next input symbol 'c' is also IDENTIFIER. Now, the top of stack matches the next input (condition B), we pop the topmost element on the stack (IDENTIFIER) and advance the input pointer.

Stack **Remaining Input**

$	SEMI_COLON	c_expression	EQ_TO_OP

= i + 5 ; $

In this configuration, the top of the stack is EQ_TO_OP. The next input symbol is =, which gets translated to EQ_TO_OP by the lexical analyser. Now, the top of the stack matches the next input (condition B), we pop the topmost element on the stack (EQ_TO_OP) and advance the input pointer.

Stack **Remaining Input**

$	SEMI_COLON	c_expression

i + 5 ; $

In this configuration, the top of the stack is a non-terminal c_expression. The next input symbol is 'i', which gets translated to IDENTIFIER by the lexical analyser. Since the top of the stack is a non-terminal (condition C), we check for the entry in the parsing table M[c_expression][IDENTIFIER]. This shows a production `c_expression → IDENTIFIER c_expr_rest`. We pop c_expression from the stack and push the right-hand side of the production in the reverse order. Observe that the input pointer is not advanced.

Stack **Remaining Input**

$	SEMI_COLON	c_expr_rest	IDENTIFIER

i + 5 ; $

In this configuration, the top of the stack is IDENTIFIER. The next input symbol 'i' is also IDENTIFIER (condition B). Now, the top of stack matches the next input (condition B), we pop the IDENTIFIER from the stack and advance the input pointer.

Stack **Remaining Input**

$	SEMI_COLON	c_expr_rest

+ 5 ; $

At this point, the top of the stack is the non-terminal c_expr_rest and the next input symbol is '+', which gets translated to OPERATOR. Since the top of the stack is a non-terminal (condition C), we check for the entry in the parsing table, M[c_expr_rest][OPERATOR]. This shows a production `c_expr_rest` → `OPERATOR c_expr_factor`. We pop the c_expr_rest and push the right-hand side of the production in the reverse order. Again, the input pointer is not advanced in this operation.

Stack **Remaining Input**

$	SEMI_COLON	c_expr_factor	OPERATOR

+ 5 ; $

In this configuration, the top of the stack is OPERATOR. The next input symbol '+' is also OPERATOR. Now, the top of the stack matches the next input (condition B), we pop the OPERATOR from the stack and advance the input pointer.

Stack **Remaining Input**

$	SEMI_COLON	c_expr_factor

5 ; $

At this point, the top of the stack is the non-terminal c_expr_factor and the next input symbol is '5', which gets translated to CONSTANT. Since the top of the stack is a non-terminal (condition C), we consult the entry in the parsing table M[c_expr_factor][CONSTANT]. This shows a production `c_expr_factor` →`CONSTANT c_expr_rest`. We pop the c_expr_factor and push the right-hand side of the production in the reverse order. Again, the input pointer is not advanced in this operation.

Stack **Remaining Input**

$	SEMI_COLON	c_expr_rest	CONSTANT

5 ; $

In this configuration, the top of the stack is CONSTANT. The next input symbol '5' is also CONSTANT. Now, the top of the stack matches the next input (condition B), we pop the CONSTANT from the stack and advance the input pointer.

Stack **Remaining Input**

$	SEMI_COLON	c_expr_rest

; $

At this point, the top of the stack is the non-terminal c_expr_rest and the next input symbol is SEMI_COLON. Since the top of the stack is a non-terminal (condition C), we check for the entry in the parsing table at M[c_expr_rest][SEMI_COLON]. This shows a production `c_expr_rest` → ε. We pop the c_expr_rest and push the right-hand side of the production in the reverse order. For an epsilon production, there is no right-hand side. Again, the input pointer is not advanced in this operation.

Stack	**Remaining Input**
$ | SEMI_COLON	; $

In this configuration, the top of the stack is SEMI_COLON. The next input symbol is also SEMI_COLON. Now, the top of the stack matches the next input (condition B), we pop the SEMI_COLON from the stack and advance the input pointer.

Stack	**Remaining Input**
$	$

Now the top of the stack is $ and the next input is also $ (condition A), which signals ***accepting the input*** as a part of the language and the successful completion of parse.

The above steps are summarised in Table 3.12.

Table 3.12 *Moves of a table-driven predictive parser program*

Stack	Input	Output/Comment
$ c_statement	`c = i + 5 ; $`	c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
$ SEMI_COLON c_expression EQ_TO_OP IDENTIFIER	`c = i + 5 ; $`	c is an Identifier
$ SEMI_COLON c_expression EQ_TO_OP	`= i + 5 ; $`	= is EQ_TO_OP
$ SEMI_COLON c_expression	`i + 5 ; $`	c_expression → IDENTIFIER c_expr_rest
$ SEMI_COLON c_expr_rest IDENTIFIER	`i + 5 ; $`	i is an Identifier
$ SEMI_COLON c_expr_rest	`+ 5 ; $`	c_expr_rest → OPERATOR c_expr_factor SEMI_COLON
$ SEMI_COLON c_expr_factor OPERATOR	`+ 5 ; $`	+ is an Operator
$ SEMI_COLON c_expr_factor	`5 ; $`	c_expr_factor → CONSTANT c_expr_rest
$ SEMI_COLON c_expr_rest CONSTANT	`5 ; $`	5 is a CONSTANT
$ SEMI_COLON c_expr_rest	`; $`	c_expr_rest → ε
$ SEMI_COLON	`; $`	; is SEMI_COLON
$	`$`	**Success**. Top of the Stack is = $

Before we end this section, let's briefly touch upon the advantages and disadvantages of table-driven predictive parsing:

Advantages

- It is easy to *generate* a table-driven parser from a given grammar. The parsing program is independent of the grammar and remains common to any grammar. The parsing table is the only component that depends on the grammar and can be generated by using the FIRST and FOLLOW set generation algorithms discussed in the next section.
- The error recovery and reporting in table-driven parser can be done easily by having entries in the table, which point to the error recovery and reporting routines.

Disadvantages

- A predictive parser like table-driven parsers or recursive descent parsers can work on LL(1) grammars only. Sometimes any amount of left-factoring and elimination of left-recursion might not be adequate to transform a grammar into LL(1) grammar.

3.4.2.1 *Example 3—A Table-driven Predictive Parser for Simple C Statement* This section shows the implementation of a table-driven parser for checking the syntax of simple C statements. This uses the grammar from Table 3.5, and the corresponding parsing table given in Table 3.11. The following dialog shows how we compile and execute the C statement checker.

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the Lexical Analyzer
$ gcc -c -o c-stmt-lex.o c-stmt-lex.c

# Building ex3 Binary
$ g++ -g -Wall ex3.cc table_parse.cc c-stmt-lex.o -o ex3

# Variant 1
$ ./ex3 'count=5;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=5;
SYNTAX CORRECT

# Variant 2
$ ./ex3 'count=index;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
count=index;
SYNTAX CORRECT

# Variant 3
$ ./ex3 'count=3*2;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=3*2;
SYNTAX CORRECT

# Variant 4
$ ./ex3 'count=count+index;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
count=count+index;
SYNTAX CORRECT
```

```
# Variant 5
$ ./ex3 'count=count+1;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=count+1;
SYNTAX CORRECT

# Variant 6
$ ./ex3 'count=count*2+5;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> EPSILON
count=count*2+5;
SYNTAX CORRECT

# Variant 7
$ ./ex3 'count=count*2+index;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
c_expr_factor -> IDENTIFIER c_expr_rest
c_expr_rest -> EPSILON
count=count*2+index;
SYNTAX CORRECT

# Missing Identifier / Constant
$ ./ex3 'count=5+;'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> CONSTANT c_expr_rest
c_expr_rest -> OPERATOR c_expr_factor
Error
SYNTAX INCORRECT

# Missing semicolon
$ ./ex3 'count=index'
c_statement -> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression -> IDENTIFIER c_expr_rest
Error
SYNTAX INCORRECT
```

3.4.2.2 *Parsing Table Entries*

In the last section, we saw how a table-driven parser can parse a given input and check for conformance with the grammar using the parsing table M. The parsing table is the only component in the table-driven parser that is dependent on the grammar. This section looks at devising a method by which we can automatically generate the parsing table entries from the grammar. This would help us in automatically generating a table-driven parser from a given grammar.

Before we look at generating the parsing table entries, let's get to some important definitions.

3.4.2.2.1 FIRST and FOLLOW Sets

A ***FIRST set*** of a non-terminal A is the set of all the terminals that A can begin with. From the grammar of Table 3.5, by visual inspection of production 1, we can figure out that the c_statement definitely begins with IDENTIFIER for a legal input. Hence, the FIRST set of c_statement is an IDENTIFIER. This is denoted by

FIRST(c_statement) = {IDENTIFIER}.

Again, by visual inspection of productions 2 and 3 of the grammar in Table 3.5, we can make out that the *c_expression* definitely begins with CONSTANT in the case of production 2 and begins with IDENTIFIER in the case of production 3. Hence, the FIRST set of c_expression consists of CONSTANT and IDENTIFIER.

FIRST(c_expression) = {CONSTANT, IDENTIFIER}.

In order to compute the FIRST sets of various symbols of a grammar in a program, we use the following rules:

Table 3.13 *Rules for computing FIRST set*

Rule #	Description
1	If X is a Terminal, then FIRST(X) = {X}
2	For a non-terminal X, If there exists a production X $\rightarrow \varepsilon$, then add ε to FIRST (X)
3	For a production $X \rightarrow Y_1\ Y_2\ Y_3\ Y_4 .. Y_k ...... Y_n$ FIRST (X)= FIRST (Y_1), if FIRST(Y_1) does not contain ε = FIRST (Y_2), if FIRST(Y_1) contains ε and FIRST(Y_2) does not contain ε = FIRST (Y_3), if FIRST(Y_1), FIRST(Y_2) both contain ε FIRST(Y_3) does not contain ε Generalising ***FIRST(X) = FIRST(Y_k)*** if FIRST(Y_1), FIRST(Y_2)..FIRST(Y_{k-1}) all contain ε and FIRST (Y_k) does not contain ε

Table 3.14 shows the computation of FIRST set for all the non-terminals in the grammar of Table 3.5. We use the rules given in Table 3.13 to compute the FIRST sets. We take each of the production and compute the FIRST set of the non-terminal in the LHS of the production.

Table 3.14 *Computation of FIRST sets*

Production	**Comments**
c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON	FIRST(IDENTIFIER) = {IDENTIFIER}, since IDENTIFIER is a non-terminal, by Rule 1 FIRST(c_statement) = FIRST(IDENTIFIER), since FIRST(IDENTIFIER) does not contain ∊ as specified by Rule 3. Hence, we add IDENTIFIER to the FIRST(c_statement) **FIRST(c_statement) = {IDENTIFIER}**
c_expression → CONSTANT c_expr_rest	FIRST(CONSTANT) = {CONSTANT}, since CONSTANT is a non-terminal, by Rule 1 FIRST(c_expression) = FIRST(CONSTANT), since FIRST(CONSTANT) does not contain ∊ as specified by Rule 3. Hence, we add CONSTANT to the FIRST(c_expression) **FIRST(c_expression) = {CONSTANT}**
c_expression → IDENTIFIER c_expr_rest	FIRST(IDENTIFIER) = {IDENTIFIER}, since IDENTIFIER is a non-terminal, by Rule 1 FIRST(c_expression) = FIRST(IDENTIFIER), since FIRST(IDENTIFIER) does not contain ∊ as specified by Rule 3. Hence, we add IDENTIFIER to the FIRST(c_expression) **FIRST(c_ expression) = {CONSTANT, IDENTIFIER}**
c_expr_rest → OPERATOR c_expr_factor	FIRST(OPERATOR) = {OPERATOR}, since OPERATOR is a non-terminal, by Rule 1 FIRST(c_expr_rest) = FIRST(OPERATOR), since FIRST(OPERATOR) does not contain ∊ as specified by Rule 3. Hence, we add OPERATOR to the FIRST(c_expr_rest) **FIRST(c_expr_rest) = {OPERATOR}**
c_expr_rest → ∊	By Rule 2, we add ∊ to the FIRST(c_expr_rest) **FIRST(c_expr_rest) = {OPERATOR, ∊}**
c_expr_factor → IDENTIFIER c_expr_rest	FIRST(IDENTIFIER) = {IDENTIFIER}, since IDENTIFIER is a non-terminal, by Rule 1 FIRST(c_expr_factor) = FIRST(IDENTIFIER), since FIRST(IDENTIFIER) does not contain ∊ as specified by Rule 3. Hence, we add IDENTIFIER to the FIRST(c_expr_factor) **FIRST(c_expr_factor) = {IDENTIFIER}**

`c_expr_factor → CONSTANT c_expr_rest`	`FIRST(CONSTANT) = {CONSTANT}, since CONSTANT is a non-terminal, by Rule 1`
	
	`FIRST(c_expr_factor) = FIRST(CONSTANT), since FIRST(CONSTANT) does not contain ∊ as specified by Rule 3.`
	
	`Hence, we add CONSTANT to the FIRST(c_expr_factor)`
	
	`FIRST(c_expr_factor) = {IDENTIFIER,CONSTANT}`

Summarising the FIRST sets of all the non-terminals from the above, we have

FIRST (c_statement)	=	{IDENTIFIER}
FIRST (c_expression)	=	{CONSTANT, IDENTIFIER}
FIRST(c_expr_rest)	=	{OPERATOR, ε}
FIRST (c_expr_factor)	=	{IDENTIFIER, CONSTANT}

The grammar in Table 3.5 is simple and does not have many productions with the same LHS. In most of the programming languages, there are quite a number of productions with the same LHS and different RHS. Due to this, there is interdependency of FIRST set of common LHS symbols with FIRST set of various RHS symbols due to the application of rule 3 on different productions. Hence the FIRST set computation needs to be done in multiple passes on the grammar rules, until a steady state is reached.

A ***FOLLOW set*** of a non-terminal A is the set of all the terminals that can follow A. From the grammar of Table 3.5, by visual inspection of production 1, we can figure out that the c_expression is definitely followed by a SEMI_COLON for a legal input. Hence, The FOLLOW set of c_expression is a SEMI_COLON. This is denoted by

FOLLOW(c_expression) = {SEMI_COLON}

The FOLLOW set would never contain ε, since it is not a valid input token. This is in contrast from the FIRST set, which can contain an ε.

In order to compute the FOLLOW sets of various symbols of a grammar in a program, we use the following rules.

Table 3.15 *Rules for computing FOLLOW set*

Rule#	Description
1.	FOLLOW(S)=$, where S is the start symbol and $ is the symbol to indicate the end of input.
2.	For a production $X \rightarrow Y_1\ Y_2\ Y_3\ Y_4 .. Y_k ...... Y_n$ where Y_1 is a non-terminal, FOLLOW(Y_1)=FIRST(Y_2), if FIRST(Y_2) does not contain ε = [union of FIRST(Y_2), and FIRST(Y_3)], if FIRST(Y_2) contains ε and FIRST(Y_3) does not contain ε. When the union of FIRST(Y_2), and FIRST(Y_3) is computed we discard the ε element out of the set. = [union of FIRST(Y_2), FIRST(Y_3) and FIRST(Y_4)], if FIRST(Y_2), FIRST(Y_3) both contain ε and FIRST(Y_4) does not contain ε. For the computation of union of FIRST(Y_2), FIRST(Y_3) and FIRST(Y_4), we discard the ε element. Generalizing, for a non-terminal Y_k in the above production ***FOLLOW(Y_k) = [union of FIRST(Y_{k+1}), FIRST(Y_{k+2})..FIRST(Y_{k+m})],*** if FIRST(Y_{k+1}), FIRST(Y_{k+2}) ..FIRST(Y_{k+3}) all contain ε and FIRST(Y_{k+m}) does not contain ε. Eventhough FIRST(Y_{k+1}), FIRST(Y_{k+2}) .. FIRST (Y_{k+m+1}) all contain ε, when we compute the union of FIRST(Y_{k+1}), FIRST(Y_{k+2})..FIRST(Y_{k+m}), we discard the ε element.

3.	For a Production $X \rightarrow Y_1\ Y_2\ Y_3\ Y_4 .. Y_k Y_{k+1}\ Y_{k+2}\ Y_{k+3} .... Y_n$ if FIRST(Y_{k+1}), FIRST(Y_{k+2}), FIRST(Y_{k+3})…FIRST(Y_n) all contain ε then FOLLOW(Y_k) = FOLLOW (X) A specific case of the rule is FOLLOW(Y_n) = FOLLOW (X) In the above type of scenarios, typically there would be another rule in the grammar of language that would allow the computation of FOLLOW (X).

Table 3.16 shows the computation of FOLLOW sets for all the non-terminals in the grammar of Table 3.5. We use the rules given in Table 3.15 to compute the FOLLOW sets. We take each of the production and compute the FOLLOW sets of the relevant symbols.

Table 3.16 *Computation of FOLLOW sets*

Production	**Comments**
c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON	**FOLLOW(c_statement) = {$}** since c_statement is the start symbol (by Rule 1) FOLLOW(c_expression) = FIRST(SEMI_COLON), by Rule 2. We know that FIRST(SEMI_COLON)= {SEMI_COLON}, since SEMI_COLON is an NonTerminal, during the computation of FIRST sets. Hence, we add SEMI_COLON to the FOLLOW(c_expression) **FOLLOW(c_expression) = {SEMI_COLON}**
c_expression → CONSTANT c_expr_rest	FOLLOW(c_expr_rest)= FOLLOW(c_expression) by Rule 3. Hence, **FOLLOW(c_expr_rest) = {SEMI_COLON}**
c_expression → IDENTIFIER c_expr_rest	FOLLOW(c_expr_rest)= FOLLOW(c_expression) by Rule 3. Hence, **FOLLOW(c_expr_rest) = {SEMI_COLON}**
c_expr_rest → OPERATOR c_expr_factor	FOLLOW(c_expr_factor)= FOLLOW(c_expr_rest) by Rule 3. Hence, **FOLLOW(c_expr_factor) = {SEMI_COLON}**
c_expr_factor → IDENTIFIER c_expr_rest	FOLLOW(c_expr_rest)= FOLLOW(c_expr_factor) by Rule 3. Hence, **FOLLOW(c_expr_rest) = {SEMI_COLON}**
c_expr_factor → CONSTANT c_expr_rest	FOLLOW(c_expr_rest)= FOLLOW(c_expr_factor) by Rule 3. Hence, **FOLLOW(c_expr_rest) = {SEMI_COLON}**

Summarising the FOLLOW sets of all the non-terminals, we have

FOLLOW (c_statement)	=	{$}
FOLLOW (c_expression)	=	{SEMI_COLON}
FOLLOW (c_expr_rest)	=	{SEMI_COLON}
FOLLOW (c_expr_factor)	=	{SEMI_COLON}

The grammar in Table 3.5 is simple and does not have many productions with the same LHS. In most of the programming languages, there are quite a number of productions with the same LHS and different RHS. Due to this, there is interdependency of FOLLOW set of common LHS symbols with FOLLOW set of various RHS symbols due to the application of rule 3 on different productions. Hence, the FOLLOW set computation needs to be done in multiple passes on the grammar rules, until a steady state is reached.

Section 3.4.2.2.2 shows a program, that computes 'FIRST' and 'FOLLOW' sets, given the grammar of the language. The FIRST and FOLLOW sets play an important role in making the parsing table entries for the top-down parsing.

3.4.2.2.2 Example 4—A Program for computing FIRST and FOLLOW sets This example shows how we can compute FIRST and FOLLOW sets for a symbol, given the grammar for the language. The following dialog shows how the binary is built and used to compute FIRST and FOLLOW sets for a given grammar.

```
# Building ex4 Binary
$ g++ -g -Wall grammar.cc ex4.cc -o ex4

# A sample grammar file - sample1.gram
$ cat sample1.gram
c_statement : IDENTIFIER EQ_TO_OP c_expression ;
c_expression : CONSTANT c_expr_rest
c_expression : IDENTIFIER c_expr_rest
c_expr_rest : OPERATOR c_expr_factor
c_expr_rest : epsilon
c_expr_factor : IDENTIFIER c_expr_rest
c_expr_factor : CONSTANT c_expr_rest


$ ./ex4 'sample1.gram'
c_expr_factor FIRST={CONSTANT,IDENTIFIER} FOLLOW={;}
c_expr_rest FIRST={OPERATOR,epsilon} FOLLOW={;}
c_expression FIRST={CONSTANT,IDENTIFIER} FOLLOW={;}
c_statement FIRST={IDENTIFIER} FOLLOW={$}

# A sample grammar file - sample2.gram
$ cat sample2.gram
E : T EDASH
EDASH : PLUS T EDASH
EDASH : epsilon
T : F TDASH
TDASH : STAR F TDASH
TDASH : epsilon
F : ( E)
F : ID

$ ./ex4 'sample2.gram'
E FIRST={(,ID} FOLLOW={$,)}
EDASH FIRST={PLUS,epsilon} FOLLOW={$,),PLUS}
F FIRST={(,ID} FOLLOW={$,),PLUS,STAR}
T FIRST={(,ID} FOLLOW={$,),PLUS}
TDASH FIRST={STAR,epsilon} FOLLOW={$,),PLUS,STAR}

# A sample grammar file - sample3.gram
$ cat sample3.gram
S : i E t S S1
S : a
S1 : e S
S1 : epsilon
E : b

$ ./ex4 'sample3.gram'
E FIRST={b} FOLLOW={t}
S FIRST={a,i} FOLLOW={$,e,t}
S1 FIRST={e,epsilon} FOLLOW={$,e,t}
```

3.4.2.2.3 Construction of Predictive Parsing Tables using FIRST and FOLLOW Sets In this section, we study about making entries in a parsing table M, using the concept of FIRST and FOLLOW sets.

Consider a production $X \rightarrow Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$ in the grammar. The predictive parsing table entries M relevant to this production are based on FIRST set of the entire RHS ($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$). The FIRST set of ($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$) is calculated as follows. Add to FIRST($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$) all the non-ε symbols of FIRST(Y_1). If FIRST(Y_1) contains ε, then add to FIRST($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$) all the non-ε symbols of FIRST(Y_2). If FIRST(Y_2) contains ε, then add to FIRST($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$) all the non-ε symbols of FIRST(Y_3). This goes on until we find a FIRST(Y_k) that does not contain ε. In case, if FIRST(Y_1), FIRST(Y_2), FIRST(Y_3) .. FIRST(Y_n) all contain ε, we add ε at the end to FIRST($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$).

For each production $X \rightarrow Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$ in the grammar, calculate the FIRST($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......Y_n$) using the above method. For each terminal 'a' in FIRST ($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$) make a parsing table entry M[X, a] = $X\rightarrow Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$. This essentially means that if we are expanding a non-terminal X and an input of 'a' is received, we use the production $X \rightarrow Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$. In case FIRST($Y_1\, Y_2\, Y_3\, Y_4..Y_k.....\, Y_n$) contains ε, we need to add more entries. The additional entries are as follows. For each terminal 'b' in FOLLOW($Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$) make a parsing table entry M[X, b] as $X \rightarrow Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$. This essentially means that if we are expanding a non-terminal X and an input of 'b' is encountered, then also we use the production $X\rightarrow Y_1\, Y_2\, Y_3\, Y_4..\, Y_k......\, Y_n$. This method used for making entries in the parsing table is summarised in Algorithm 3.2.

```
Grammar G
α is a string of grammar symbols e.g. Y1 Y2 Y3 Y4.. Yk..... Yn
```

```
Initialise the table M to undefined entries
for (each production A → α in Grammar G){
     for ( each terminal a in FIRST (α) ){
          M[A,a] = (A → α)
     }
     if (FIRST(α) contains ε){/* additional entries */
          for ( each terminal b in FOLLOW(A)){
               M[A,b] = (A →α)
          }
          if ( $ is in FOLLOW(A)) {
               M[A,$] = ( A →α)
          }
     }
}
```

Algorithm 3.2 *Construction of predictive parsing table*

We now use Algorithm 3.2 to make the parsing table entries for the grammar described in Table 3.5. We use the FIRST and FOLLOW sets computed earlier as a ready reference.

Production	Comments
(1) c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON	FIRST(IDENTIFIER EQ_TO_OP c_expression SEMI_COLON) = FIRST(IDENTIFIER) = {IDENTIFIER} Hence, **M[c_statement, IDENTIFIER] = 1 /* production 1 */**
(2) c_expression → CONSTANT c_expr_rest	FIRST(CONSTANT c_expr_rest)= FIRST(CONSTANT) = { CONSTANT} Hence, **M[c_expression, CONSTANT] = 2 /* production 2 */**
(3) c_expression → IDENTIFIER c_expr_rest	FIRST(IDENTIFIER c_expr_rest)= FIRST(IDENTIFIER) = IDENTIFIER Hence, **M[c_expression, IDENTIFIER] = 3 /* production 3 */**
(4) c_expr_rest → OPERATOR c_expr_factor	FIRST(OPERATOR c_expr_factor)=FIRST(OPERATOR) = {OPERATOR} Hence, **M[c_expr_rest,OPERATOR] = 4 /* production 4 */**
(5) c_expr_rest → ∈	FIRST(∈) = {∈} FOLLOW(c_expr_rest)= {SEMI_COLON} from the earlier computation. Hence, **M[c_expr_rest, SEMI_COLON]= 5 /* production 5 */**
(6) c_expr_factor → IDENTIFIER c_expr_rest	FIRST(IDENTIFIER c_expr_rest) = FIRST(IDENTIFIER) = {IDENTIFIER}, **M[c_expr_factor,IDENTIFIER]= 6 /* production 6 */**
(7) c_expr_factor → CONSTANT c_expr_rest	FIRST(CONSTANT c_expr_rest) = FIRST(CONSTANT) = {CONSTANT}, **M[c_expr_factor, CONSTANT]= 7 /* production 7 */**

The reader is advised to verify the parse table entries made above and compare it with the parse table shown in Table 3.11 for explaining the predictive table-driven parser algorithm.

3.4.2.3 Example 5—Predictive Parsing Table Entries using FIRST and FOLLOW Sets This example shows how the parsing table entries of a table-driven predictive parser are generated using FIRST and FOLLOW set concepts. Algorithm 3.2 has been implemented in this example. This example not only generates the parsing table entries, but also does syntax analysis of the simple c assignment statement using the parse table entries.

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the Lexical Analyzer
$ gcc  -c -o c-stmt-lex.o c-stmt-lex.c

# Building ex5 Binary
$ g++ -g -Wall grammar.cc table_parse.cc c-stmt-lex.o ex5.cc -o ex5

# Variant 1
$ ./ex5 'sample1.gram'  'count=5;'

**** Table Entries ****
M[c_statement][IDENTIFIER]= 'c_statement : IDENTIFIER EQ_TO_OP c_expression ; '
M[c_expression][CONSTANT]= 'c_expression : CONSTANT c_expr_rest  '
M[c_expression][IDENTIFIER]= 'c_expression : IDENTIFIER c_expr_rest  '
M[c_expr_rest][OPERATOR]= 'c_expr_rest : OPERATOR c_expr_factor  '
M[c_expr_rest][;]= 'c_expr_rest : epsilon  '
M[c_expr_factor][IDENTIFIER]= 'c_expr_factor : IDENTIFIER c_expr_rest  '
M[c_expr_factor][CONSTANT]= 'c_expr_factor : CONSTANT c_expr_rest  '

**** Parsing ****
c_statement : IDENTIFIER EQ_TO_OP c_expression ;
c_expression : CONSTANT c_expr_rest
c_expr_rest : epsilon
count=5;
SYNTAX CORRECT

# Variant 2
$ ./ex5 'sample1.gram'  'count=index;'

**** Table Entries ****
M[c_statement][IDENTIFIER]= 'c_statement : IDENTIFIER EQ_TO_OP c_expression ;  '
M[c_expression][CONSTANT]= 'c_expression : CONSTANT c_expr_rest  '
M[c_expression][IDENTIFIER]= 'c_expression : IDENTIFIER c_expr_rest  '
M[c_expr_rest][OPERATOR]= 'c_expr_rest : OPERATOR c_expr_factor  '
M[c_expr_rest][;]= 'c_expr_rest : epsilon  '
M[c_expr_factor][IDENTIFIER]= 'c_expr_factor : IDENTIFIER c_expr_rest  '
M[c_expr_factor][CONSTANT]= 'c_expr_factor : CONSTANT c_expr_rest  '

**** Parsing ****
c_statement : IDENTIFIER EQ_TO_OP c_expression ;
```

```
c_expression : IDENTIFIER c_expr_rest
c_expr_rest : epsilon
count=index;
SYNTAX CORRECT
```

3.4.2.4 *Error Recovery in Predictive Parsing*

We had, till now, discussed about the two common techniques of implementing a top-down predictive parser, viz. recursive descent parsing and table-driven parsing. This section is focussed on the error reporting and recovery in top-down predictive parsing.

In table-driven parsing, it is clear as to what terminals and non-terminals the parser expects from the rest of the input. An error can be detected in the following situations:

1. When the terminal on the top of the stack does not match the next input symbol.
2. When a non-terminal A is on the top of stack, *a* is the next input symbol and the parsing table entry M[A,a] is empty.

Recall from Section 3.4 that the error recovery of a parser is the ability to ignore the current error and continue with the parsing for the remainder of the input. The error recovery schemes that are commonly used in predictive parsing are:

1. Panic mode recovery.
2. Phrase level recovery.

Panic mode recovery is based on the principle that when an error is detected, the parser should skip the input symbols until it finds a synchronising token in the input. Usually, the synchronising tokens are more than one; hence a set called as ***synchronising set*** is used to denote the set of all synchronising tokens. The effectiveness of panic mode recovery depends on the choice of synchronising set. Some of the guidelines for constructing the synchronising set are as follows:

1. For a non-terminal A, all the elements of FOLLOW set of A can be added to synchronising set of A. For example, consider an input of multiple C statements as shown below to be verified against the grammar in Table 3.5. The line 1 has a missing C expression.

   ```
   count = ; /* This is line 1 – Missing C Expression*/
   index = a +100 ; /* This is line 2 */
   ```

   The table-driven parser after consuming = in line 1, would expect a C expression, instead it would find an input of SEMI_COLON . This is a part of FOLLOW set of C expression. The parser can emit a message indicating that C expression is missing and then continue parsing the line 2.
2. For a non-terminal A, the elements in FIRST set of A can be added to synchronisation set of A. This would be useful in situations where the parsing can be resumed according to A, on appearance of an input symbol that is a part of FIRST set of A. For example, consider an erroneous C statement using the grammar in Table 3.5

   ```
   39 count = a + b ; /* Extra characters 39 */
   ```

   The table-driven predictive parser can skip characters '39' and synchronise from count (Identifier), because the FIRST (c_statement) is an IDENTIFIER. This would allow the parser to recover from errors in which there are some extraneous characters and resume the parsing from 'count'.
3. The synchronising set for a token can be a set of all other tokens. If a terminal on the top of stack cannot be matched, then pop the terminal from the top of stack and issue a warning indicating that the terminal was inserted and continue parsing. For example, consider an input C statement using the grammar in Table 3.5 as shown below.

   ```
   count 45 ; /* Missing equal to operator */
   ```

   The table-driven parser after consuming count would expect an EQ_TO_OP according to rule 1. On finding that the next input symbol is 45 (which is a CONSTANT), the table-driven predictive parser can emit a message indicating that EQ_TO_OP was inserted and continue parsing as if EQ_TO_OP was a part of the input.

Phrase-level recovery in predictive parser can be implemented by filling in blank entries in the predictive parsing table with pointers to error-handling routines. For example, in Table 3.11 each one of the empty entries can point to error-handling routines that can do the following:

1. The error-handling routines can insert, modify or delete any symbols in the input. This would amount to local correction that is expected from phrase-level recovery.
2. The routines can also pop elements from the stack. The routines should safeguard against an infinite loop by making sure whatever local correction is done should ultimately result in an input symbol being consumed.

3.5 BOTTOM-UP PARSING

In bottom-up parsing, the parse tree for an input string is constructed beginning at the leaves (the bottom) and working up towards the root (the top).

Bottom-up parsing involves 'reducing' an input string 'w' to the start symbol of the grammar. In each of the reduction step a particular sub-string matching the right side of a production is replaced by a symbol on the left of that production. If the sub-string were chosen correctly at each step, the reduction steps would be the exact reverse of rightmost derivation. The 'reducing' is in contrast to what we saw earlier in top-down parsing where we were 'expanding' the left-hand-side of the production to replace it with the RHS of the production.

Let us consider the example of a C statement '*count* = *index* + *10*; We shall try to check if this C statement is in conformance to the grammar in Table 3.1 (which is where we started before we made changes in grammar to suit LL parsing). It is reproduced below for convenience.

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression	→	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR c_expression

For the input '*count* = *index* + *10*;' the lexical analyser returns the following string of tokens by virtue of the definition of an IDENTIFIER, CONSTANT, OPERATOR and EQ_TO_OP. This is shown as:

```
=> IDENTIFIER  EQ_TO_OP  IDENTIFIER  OPERATOR  CONSTANT  SEMI_COLON              Tokens
```

We scan the tokenised input string to identify sub-strings that match the right side of a production. The IDENTIFIER qualifies for being the right side of Production 3. Let's choose to replace one of the IDENTIFIER (the one standing in for variable 'index') by the equivalent left side of Production 3, i.e. c_ expression. This results in

```
=> IDENTIFIER  EQ_TO_OP  c_expression  OPERATOR  CONSTANT  SEMI_COLON  By Production 3
```

Again, we scan the above string to identify sub-strings that match the right side of a production. The 'CONSTANT' sub-string qualifies for being the right side of Production 2. Let's choose to replace it by the equivalent left side of Production 2, i.e. c_expression. This results in the following string:

```
=> IDENTIFIER  EQ_TO_OP  c_expression  OPERATOR  c_expression  SEMI_COLON By Production 2
```

We continue the next iteration of scanning the above string to identify sub-strings that match the right side of a production. The sub-string 'c_expression OPERATOR c_expression' qualifies for being the right side of Production 4. Let's choose to replace it by the equivalent left-side of Production 4, i.e. c_expression. This results in the following string:

```
=> IDENTIFIER EQ_TO_OP c_expression SEMI_COLON                    By Production 4
```

Let's see the next iteration of scanning the above string to identify sub-strings that match the right side of a production. The entire string 'IDENTIFIER EQ_TO_OP c_expression SEMI_COLON' qualifies for being the right side of Production 1. Let's replace it by the equivalent left side of Production 1, i.e. C statement. This results in the following string:

```
=> c_statement                                                    By Production 1
```

Thus we have ***reduced*** the input string 'count = index + 10 ;' to a C statement. Interestingly, note that the derivation is exactly the reverse of a rightmost derivation starting from the root:

c_statement	=>	IDENTIFIER	EQ_TO_OP	c_expression			SEMI_COLON	*By P1*
	=>	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR	c_expression	SEMI_COLON	*By P4*
	=>	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR	CONSTANT	SEMI_COLON	*By P2*
	=>	IDENTIFIER	EQ_TO_OP	IDENTIFIER	OPERATOR	CONSTANT	SEMI_COLON	*By P3*
		count	=	index	+	10	SEMI_COLON	

The production rules applied in the rightmost derivation starting from the top are 1 followed by 4, 2, 3 while the bottom-up parsing applied productions 3 followed by 2, 4, 1. A rightmost derivation in reverse is called as ***canonical reduction sequence***.

The parse-tree construction for bottom-up parsing is illustrated in Fig. 3.7.

We saw in the previous sections that left factoring and elimination of left recursion are two important transformations essential to make a grammar suitable for top-down parsing. Similarly, the transformation that is necessary to make the grammar suitable for bottom-up parsing is elimination of right recursion. In top-down parsing, we started with the grammar in Table 3.1 and transformed it to have left factoring and eliminate left recursion. Similarly, we start with the grammar in Table 3.1 and eliminate right recursion in rule 4 to make it suitable for bottom-up parsing. Table 3.1 is repeated here for convenience.

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression	→	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR c_expression

Consider the Production 4,

c_expression → c_expression OPERATOR c_expression

The right recursion can end only in two situations, where the c_expression takes the form of CONSTANT (as in rule 2) or IDENTIFIER (as in rule 3). Taking advantage of it, we can rewrite Production 4 as two rules shown below.

c_expression → c_expression OPERATOR CONSTANT
c_expression → c_expression OPERATOR IDENTIFIER

Using the two rules above, the c-statement grammar suitable for bottom-up parsing can be written as:

Table 3.17 *C-statement grammar suitable for bottom-up parsing*

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression	→	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR CONSTANT
5		|	c_expression OPERATOR IDENTIFIER

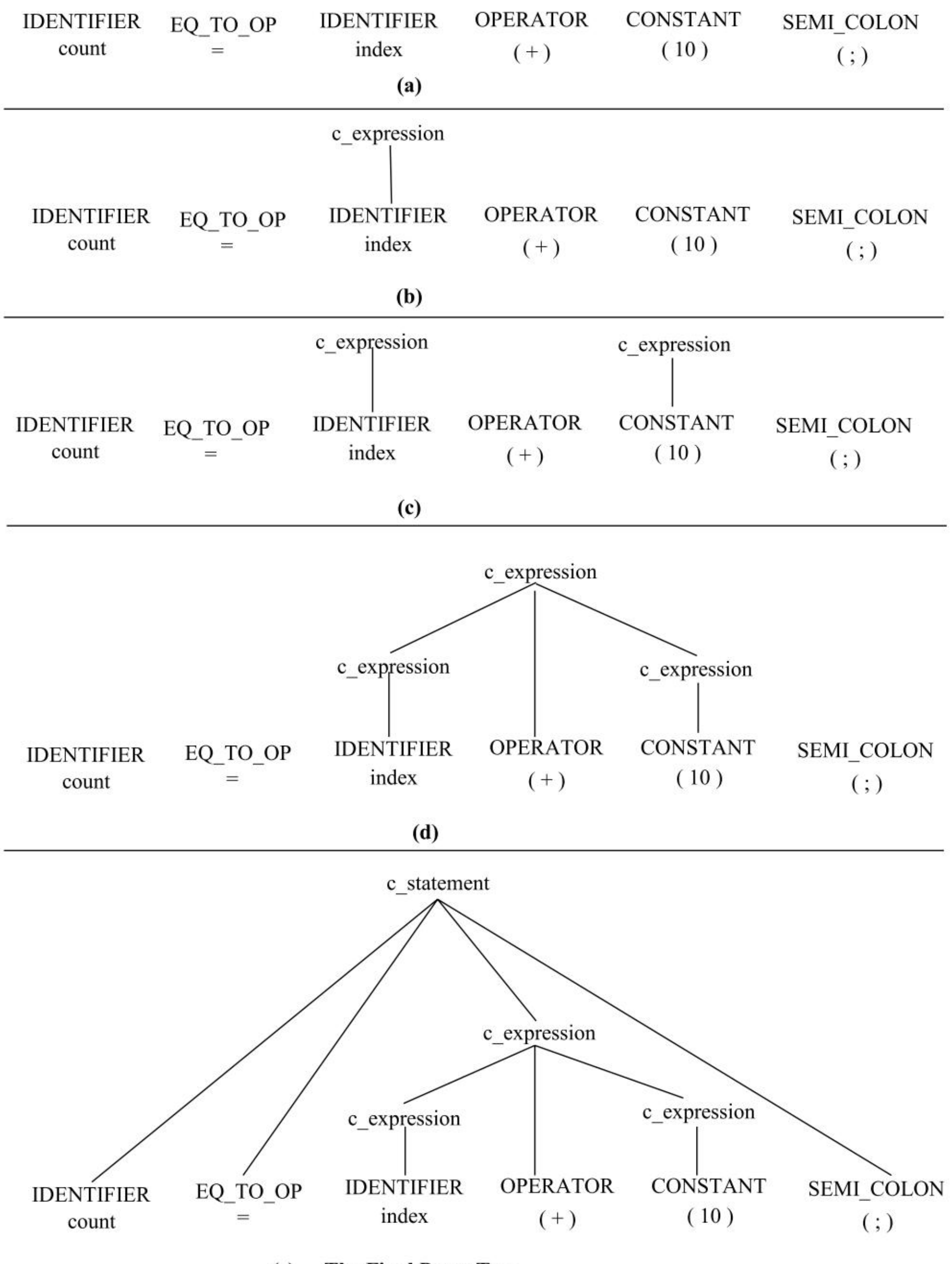
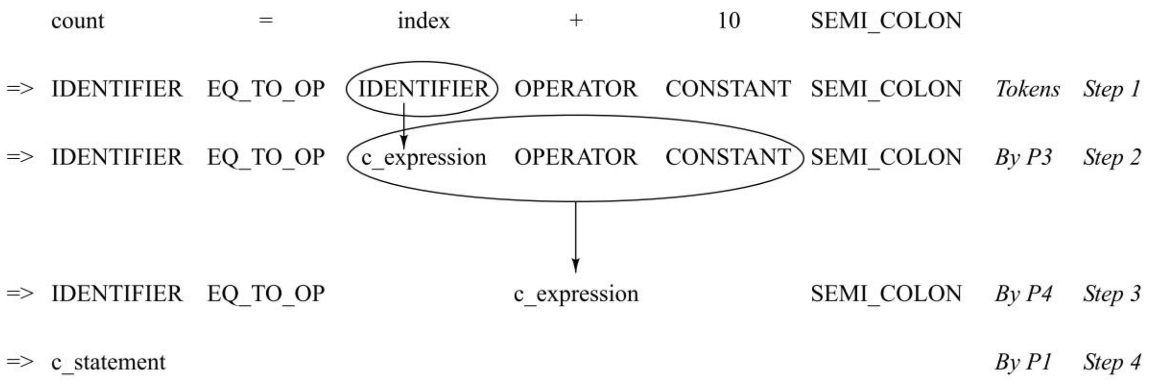


Fig. 3.7 *Bottom-up parsing for count=index + 10;*

3.5.1 Definitions in Bottom-up Parsing

Let's go through the derivation of '*count = index + 10* ;' with the new grammar rules shown in Table 3.17 and in the process understand some definitions.



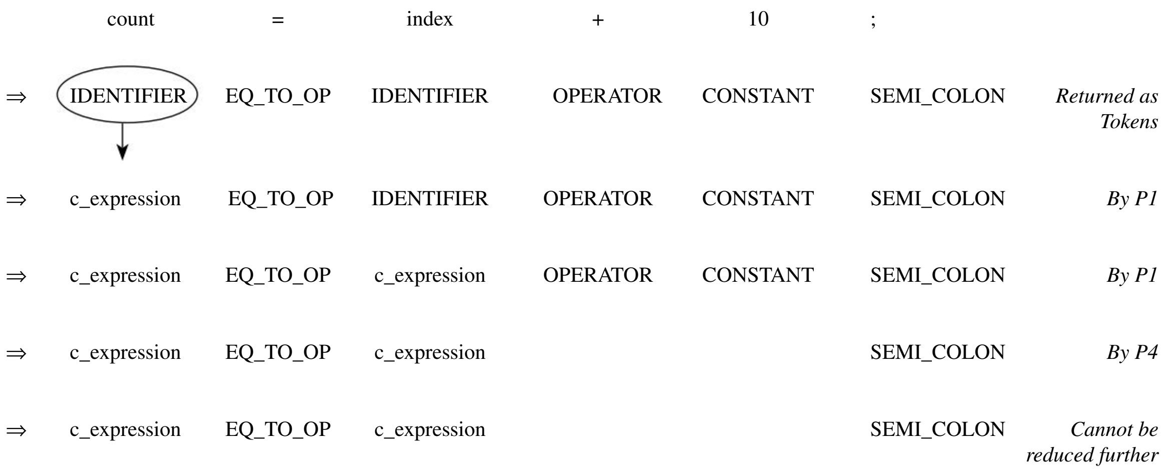
```
    count         =        index         +         10         SEMI_COLON

=>  IDENTIFIER  EQ_TO_OP  IDENTIFIER    OPERATOR  CONSTANT    SEMI_COLON   Tokens   Step 1

=>  IDENTIFIER  EQ_TO_OP  c_expression  OPERATOR  CONSTANT    SEMI_COLON   By P3    Step 2

=>  IDENTIFIER  EQ_TO_OP         c_expression                 SEMI_COLON   By P4    Step 3

=>  c_statement                                                            By P1    Step 4
```

The process of replacement of the right side of the production by its equivalent left side of the Production is called as ***reduction***. The sub-string that gets replaced by its equivalent left side of the production is called a ***handle***. The handles in two of the steps above are shown. In the above derivation, each one of the intermediate forms that are encountered before the input is reduced to the start symbol is called as ***right sentential form***. For example, in the derivation of *'count=index + 10 ;'* shown above, in step 1.

```
=> IDENTIFIER EQ_TO_OP IDENTIFIER OPERATOR CONSTANT SEMI_COLON
```

is a right sentential form whose handle is c_expression → IDENTIFIER at position 3.

In the derivation of '*count=index + 10* ;' shown above, the choices of handles to replace are intuitively clear. However, in reality the choices of handles have to be pruned to identify the correct handle that would lead us to the start symbol. For example, by making a wrong choice of handle the derivation could have gone awry as follows:



```
     count            =           index          +          10         ;

⇒   IDENTIFIER     EQ_TO_OP    IDENTIFIER     OPERATOR    CONSTANT    SEMI_COLON    Returned as
                                                                                         Tokens

⇒   c_expression   EQ_TO_OP    IDENTIFIER     OPERATOR    CONSTANT    SEMI_COLON          By P1

⇒   c_expression   EQ_TO_OP    c_expression   OPERATOR    CONSTANT    SEMI_COLON          By P1

⇒   c_expression   EQ_TO_OP    c_expression                           SEMI_COLON          By P4

⇒   c_expression   EQ_TO_OP    c_expression                           SEMI_COLON     Cannot be
                                                                                reduced further
```

The input is not reduced to c_statement, this would have led us to the wrong conclusion that given input *'count = index + 10;'* is not a valid C statement. This undermines the need for identifying the correct handle out of all possible handles (handle pruning) to replace in order to be able to reduce the input string to the start symbol.

Similarly, when there is more than 1 production that has the same right-hand side, the choice of which production rule to apply, in order to reduce a handle is very important to have the correct outcome of parsing.

3.5.2 Working of a Bottom-up Parser

Bottom-up parsing is commonly implemented by using a stack. The end of the input is marked with a '$'. The bottom of the stack is also marked with '$'. Depending on the content of the stack and the next input symbol, the actions of a bottom-up parser can be classified as (1) shift (2) reduce (3) accept (4) error.

In ***shift*** action, the next input symbol is shifted on to the top of the stack.

In ***reduce*** action, the top few elements on the stack constituting the right-hand side of a production are replaced by the non-terminal that forms the equivalent left side on the top of stack.

In ***accept*** action, in which the top of the stack contains the start symbol, the parser announces the successful parsing of the input.

In ***error*** action, the parser recognises that a syntax error has occurred in the input and calls an error handling routine.

Let's see the working of a bottom-up parser by trying to verify if an input string '*a= count +2* ;' is in conformance with the grammar given in Table 3.17.

The first input symbol 'a' would be translated to IDENTIFIER by the lexical analyser. The parser would perform the 'shift' action and push it on the stack. The stack is shown below.

$	IDENTIFIER

Step 1

The next symbol '=' would be translated to EQ_TO_OP by the lexical analyser. The parser would perform the 'shift' action and push it on the stack.

$	IDENTIFIER	EQ_TO_OP

Step 2

The next input symbol 'count' would be translated as IDENTIFIER by the lexical analyser. The parser would perform the 'shift' action and push it on the stack.

$	IDENTIFIER	EQ_TO_OP	IDENTIFIER

Step 3

The topmost element of the stack is IDENTIFIER, which forms the right-hand side of Production 3. A 'Reduce' action is performed by the parser, replacing IDENTIFIER by c_expression.

$	IDENTIFIER	EQ_TO_OP	c_expression

Step 4

The next input symbol '+' would be translated as 'OPERATOR' by lexical analyser. The parser would perform the 'shift' action and push it on the top of stack.

$	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR

Step 5

The next input symbol '2' would be translated into CONSTANT by lexical analyser. The parser would perform 'shift' action and push it on the top of stack.

$	IDENTIFIER	EQ_TO_OP	c_expression	OPERATOR	CONSTANT

Step 6

The top-most few elements of the stack are c_expression OPERATOR CONSTANT, which forms the right-hand side of Production 3. A 'reduce' action is performed by the parser, replacing 'c_expression OPERATOR CONSTANT ' by c_expression.

$	IDENTIFIER	EQ_TO_OP	c_expression

Step 7

The next input symbol ';' would be translated into SEMI_COLON by lexical analyser. The parser would perform 'shift' action and push it on the top of stack.

$	IDENTIFIER	EQ_TO_OP	c_expression	SEMI_COLON

Step 8

The top-most few elements of the stack are IDENTIFIER EQ_TO_OP c_expression SEMI_COLON, which forms the right-hand side of Production 3. A 'reduce' action is performed by the parser, replacing 'IDENTIFIER EQ_TO_OP c_expression SEMI_COLON' by c_statement.

$	c_statement

Step 9

The top of the stack contains the start symbol 'c_statement'. The parser does an accept action and indicates the **successful parsing** of the input to c_statement.

Note that the rationale behind why a shift is chosen or reduce is chosen in each one of the above steps is not yet explained. We will come to that a little later. For now, try and appreciate the how the stack is manipulated to make the entire parse.

Let's look at an example to see how the shift/reduce actions happen during the parsing process. We shall use a parser generated by the tool ***bison*** (similar to the Example 1 in Section 3.1.2). The tool bison generates a bottom-up parser that does shift and reduce, the way we did above. Let's try it out by giving in the grammar of Table 3.17 and the same input as above *'a = count+2;'*

3.5.2.1 *Example 6—A Bottom-up Parser for a Simple C Statement* The parsers that are automatically generated by most of popular tools like bison/yacc are bottom-up parsers. This section shows a bottom-up parser generated by bison, for the grammar in Table 3.17. The focus of this example is to illustrate the shift-reduce actions performed by a bottom-up parser, when an input is given. We reuse the lexical analyser shown in Example 1.

```
# Generating C File from grammar. -t enables debugging, i.e. shows shift/reduce actions performed
$ bison -dy -t -oc-stmt-bot-up-gram.c -v c-stmt-bot-up-gram.y

# Compiling the Parser
$  gcc  -g  -Wall  -DGENERATED_PARSER  -c  -o  c-stmt-bot-up-gram.o  c-stmt-bot-up-gram.c

# Generating the lexical Analyzer from lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the lexical Analyzer
$ gcc -c -DGENERATED_PARSER -DCHAP3_EX6 -o c-stmt-lex6.o c-stmt-lex.c

# Building ex6 Binary
$ gcc -g -Wall ex6.c c-stmt-bot-up-gram.o c-stmt-lex6.o -o ex6

# An Example for shift-reduce Parsing
$ ./ex6 'a=count+2;'
Starting parse
Entering state 0
Reading a token: Next token is token IDENTIFIER ()
Shifting token IDENTIFIER ()
Entering state 1
```

```
Reading a token: Next token is token EQ_TO_OP ()
Shifting token EQ_TO_OP ()
Entering state 3
Reading a token: Next token is token IDENTIFIER ()
Shifting token IDENTIFIER ()
Entering state 5
Reducing stack by rule 3 (line 21):
   $1 = token IDENTIFIER ()
-> $$ = nterm c_expression ()
Stack now 0 1 3
Entering state 7
Reading a token: Next token is token OPERATOR ()
Shifting token OPERATOR ()
Entering state 8
Reading a token: Next token is token CONSTANT ()
Shifting token CONSTANT ()
Entering state 11
Reducing stack by rule 4 (line 22):
    $1 = nterm c_expression ()
    $2 = token OPERATOR ()
    $3 = token CONSTANT ()
->  $$ = nterm c_expression ()
Stack now 0 1 3
Entering state 7
Reading a token: Next token is token SEMI_COLON ()
Shifting token SEMI_COLON ()
Entering state 9
Reducing stack by rule 1 (line 16):
    $1 = token IDENTIFIER ()
    $2 = token EQ_TO_OP ()
    $3 = nterm c_expression ()
    $4 = token SEMI_COLON ()
->  $$ = nterm c_statement ()
Stack now 0
Entering state 2
Reading a token: Now at end of input.
Stack now 0 2
Cleanup: popping nterm c_statement ()
a=count+2;
SYNTAX CORRECT
```

For now, ignore the state related information emitted in the output. The reader is advised to make a comparison of the output of *'ex6'* for *'a=count+2;'* in the above dialog with the explanation done previously so as to understand the process of shift-reduce parsing better. Trying out the executable ex6 with different types of input will help in comprehending the nuances of shift-reduce parsing.

3.5.3 Shift-Reduce Parsing Methods

From a mechanical point of view, the explanation given for shift-reduce parsing in the previous section seemed complete. Let's do a deeper analysis and find out some of the issues encountered in the parsing of input.

How do we recognise the right 'handles' for reducing?

For example, in the shift-reduce parsing of *'a=count+2;'* explained earlier. When we were in step 1, we had the stack where 'IDENTIFIER' was on the top of stack. We did not reduce it to *'c_expression'*. However, in step 3, we reduced the 'IDENTIFIER' on the top of the stack to *'c_expression'*. If we had reduced 'IDENTIFIER' on the top of the stack to *'c_expression'* in step 1, it would have led us to the conclusion that the sentence *'a=count+2;'* does not conform to the c-statement grammar.

This explains the ***need to have a mechanism to identify the correct handle for reducing***.

Similarly, in cases where there are more than 1 productions having the same sub-string on the right-hand side we need to have a mechanism to figure out ***what production needs to be used for reducing*** in order to have a successful derivation of input.

In the next few sections, we shall see two methods of shift-reduce parsing, which use different techniques to resolve the issues mentioned above. The two methods are:

1. Operator precedence parsing method.
2. LR parsing method.

These methods vary in their approach to identify the right handle and the correct production for reduction. The LR parsing method uses a state machine represented by a parsing table to make the shift/reduce decisions. The operator parsing method uses a table known as precedence table to make shift/reduce decisions.

The operator precedence parsing method can be applied to a small class of grammar called as ***operator precedence grammar***. In operator precedence grammar, there would not be an epsilon production. The operator precedence grammar also puts a restriction that in no production would there be two adjacent terminals. The operator precedence parsing method is described in Section 3.5.4.

The LR parsing method can be applied to a class of grammar called LR grammar (L stands for left to right scanning and R stands for rightmost derivation). Unlike the operator precedence grammar, the LR grammar does not put serious restrictions on the productions. The LR parsing method can parse most of the programming constructs. The LR parsing method is described in detail in Section 3.5.5.

Even when we devise mechanisms to identify the right handle for reducing and the correct production for reducing, there is no assurance that a shift-reduce parser can parse any context-free grammar. For some context-free grammars, it is possible that the shift-reduce parser can get into a configuration in which the parser knowing the entire stack contents (S) and the next input symbol (a), is still unable to decide whether to shift or reduce (***shift-reduce conflict***) or cannot decide which of the productions to reduce (***reduce-reduce conflict***). These kinds of context-free grammars for which the LR parsers end up with a conflict are called as ***non-LR grammars***.

3.5.4 Operator Precedence Parsing

Operator precedence parsing is a shift-reduce parsing method that can be applied to a small class of grammar called as ***operator grammar***.

An operator grammar has two important characteristics:

1. There are no ε productions in this type of grammar.
2. No production would have two adjacent non-terminals.

Consider an operator grammar recognising expressions using the following 7 productions.

Table 3.18 *An operator grammar*

1	E →		ID	*An expression is a Identifier*
2		|	E + E	*Addition*
3		|	E – E	*Subtraction*
4		|	E * E	*Multiplication*
5		|	E / E	*Division*
6		|	E ^ E	*Raising to the Power*
7		|	(E)	*Parenthesised*
8		|	-E	*Unary Minus*

The ID can be either a number or a variable name.

Before we see how the operator precedence parsing technique can be used to parse an input and check its conformance with the above grammar, let's spend some time on where this kind of grammar can be used.

In the first look, it appears that the grammar shown in Table 3.18 can be used in a desktop calculator, if we define ID as a number. In reality, apart from the desktop calculator, most of the programming languages use this kind of grammar for evaluating expressions. Remember, we were having a *c_expression* in our earlier examples. We can use operator precedence grammar to evaluate such expressions. The existence of expressions in almost all programming languages makes it an important class of grammar. The limitation on the type of productions is not a great hindrance, since expressions by nature exhibit operator grammar characteristics. To appreciate the importance of this parsing, it is relevant to mention that the expressions in SNOBOL compiler were based on operator precedence parsing.

We had seen earlier that in shift-reduce parsing, the main challenges are:

1. To identify the correct handle in each of the reduction steps, such that we can eventually reduce a given input to the start symbol.
2. To identify which production to use for reducing in each of the reducing steps, such that we can correctly reduce the given input to the start symbol.

The operator precedence parsing technique uses a table called as ***operator precedence relations*** table for making informed decisions with regard to identifying the correct handle during a reduction step. Since the operator grammar does not contain any epsilon productions, the issue of having to identify the right production for reducing is simplified.

An operator precedence parser consists of:

1. An input buffer that contains the string to be parsed followed by a $, a symbol used for indicating the end of input.
2. A stack containing the sequence of grammar symbols with a $ at the bottom of the stack. The stack does not differentiate between non-terminals, it merely marks the element on the stack as non-terminal. This will be more evident, when we see the example a little later.
3. An operator precedence relations table O, containing the precedence relationship between a pair of terminals. This is a two-dimensional array using the operators as the indexes. The content of the operator precedence relations table can be one of the three precedence relations namely <•, •>, or $\doteq$ between a pair of terminals. The relation ***a* •> *b*** implies that the terminal 'a' has higher precedence than 'b'. The relation ***a* <• *b*** implies that the terminal 'a' has lower precedence than 'b'. The relation ***a*** $\doteq$ ***b*** implies that both 'a' and 'b' have the same precedence. The precedence relationship between a terminal 'a' and another terminal 'b' is determined by consulting the entry O[a][b]. Table 3.19

shows an operator precedence relations table. The shaded entry indicates that operator * has a higher precedence than '–'.

4. An operator precedence parsing program that takes the input string and determines if it is conformant to the grammar. The parser program uses the operator precedence relations table and the stack to arrive at this decision.

Figure 3.8 illustrates the various components of an operator precedence parser.

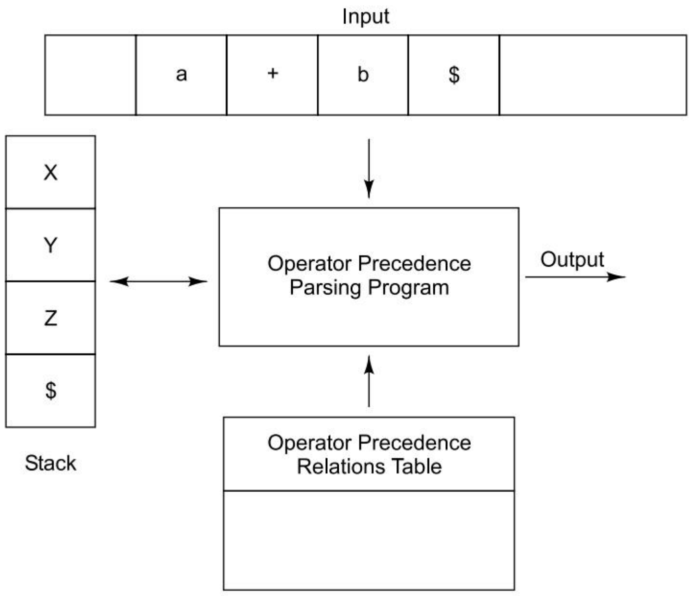

Input
a
+
b
$
X
Y
Z
$
Stack
Operator Precedence Parsing Program
Output
Operator Precedence Relations Table

Fig. 3.8 *Components of an operation precedence parser*

The operator precedence parsing program determines the action of the parser depending on

- 'a', the top most terminal symbol on the stack
- 'b', the current input symbol

There are 3 combinations of 'a' and 'b' that are important for the parsing program

Condition	Action
a = b = $	The parser announces the successful completion of parsing and returns.
a* <• *b* or *a* $\doteq$ *b	The parser shifts the input symbol on to the top of the stack and advances the input pointer to the next input symbol.
a* •> *b	This triggers off a reduce operation. The parser pops out elements one by one from the shift-reduce stack until we find that the current top of the stack element has lower precedence than the most recently popped-out terminal. The popped out elements form the handle for the reduction. We make a check to see if the handle forms a valid right-hand side of a production. If the handle is valid then we push a non-terminal on the stack and continue the parse or else we try to decipher the error and emit a proper error message. The parser can either recover and continue the parse or stop parsing.

The parsing program is summarised in Algorithm 3.1.

```
op_prec_parse()
{
      do
      {
           a = Symbol on the top of Stack
           b = Next Input Symbol

           if ( ( a == $) && ( b == $)){
                return(SUCCESS)
           }

           if ( O[a][b] is < || O[a][b] is ≐){/* Shift */
                Shift b on to the Stack
                Advance the input pointer
                b = next input symbol ;
           } else if ( O[a][b] is >) {/* Reduce */
                no_of_term_popped = 0;
                do {
                     if( top_of_stack == NONTERM ){
                          pop the stack
                          continue;
                     }

                     /* Terminal */

                     if (no_of_term_popped > 0){
                          if(O[topmost_term_on_stack][most_recently_popped_term] is <){
                                         break;
                          }
                     }
                     most_recently_popped_term = topmost_term_on_stack;
                     pop the stack
                     no_of_term_popped ++ ;

                } while ( 1)

                a = top_of_the stack
                if ( the elements popped out do not form the RHS of a Production){
                     return(FAILURE)
                }
                Push non-terminal E on the stack

           } else {
                return(FAILURE)
           }
      } while ( 1)
}
```

Algorithm 3.3 *Operator precedence parsing*

The main data structure on which the operator precedence algorithm depends is the operator precedence relations table O. For the grammar mentioned in Table 3.18, the operator precedence relations table O is shown in Table 3.19.

Table 3.19 *Operator precedence relations table*

	+	–	*	/	^	**ID**	**(**	**)**	**$**
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
ID	•>	•>	•>	•>	•>	ERR	ERR	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	≐	ERR
)	•>	•>	•>	•>	•>	ERR	ERR	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	ERR	ERR

Using the grammar shown in Table 3.18, and the corresponding parsing table given in Table 3.19, let's see how the operator precedence parser algorithm works for an input of

'80 + 100 – 56'.

Initially in operator precedence parsing, the end of input marker '$' is pushed on to the stack. '$' is the top of the stack now.

Stack	**Remaining Input**
$	80 + 100 – 56 $

The parser receives the first token '80', which is translated to an ID from the lexical analyser. The parser compares the precedence of ID with '$' by viewing the entry O[$][ID]. The value is '<•', which implies that the input symbol ID should be pushed on to the stack (shift operation) and the input pointer advanced. The topmost terminal symbol on the stack is ID.

Stack	**Remaining Input**
$ | ID	+ 100 – 56 $

The next input is '+', which is a token of type PLUS as classified by the lexical analyser. The parser compares the precedence of ID (which is the top of stack) with '+' by viewing the entry O [ID] [+]. The value is '•>', which implies that a reduce operation needs to be carried out. Symbols are popped out until the precedence relation between the top of stack element and the most recently popped element is '<•'. The first symbol to be popped out of the stack is ID. This makes the most recently popped out element as ID. The stack then has '$' as the top of the stack. The precedence relation between '$' (top of stack) and the most recently popped out element (ID) is determined by the entry O[$][ID]. The value is '<•', which signals us to stop popping out the elements from stack. The elements that were popped out during this process are ID. We reduce it to a non-terminal E. Note that we are not distinguishing between non-terminals, it is always E in the case of operator grammar. We need to push E onto the stack. The topmost terminal symbol on the stack is, however, $. Note that the input pointer is not advanced in the reduce operation.

Stack	**Remaining Input**
$ | E	+ 100 – 56 $

We continue to use ‘+’ as the next input symbol. The parser compares the precedence of $ (topmost terminal symbol on the stack) with ‘+’ with by viewing the entry O[$][+]. The value is ‘<•’, which implies that the input symbol + should be pushed on to the stack (shift operation) and the input pointer advanced. The topmost terminal symbol on the stack is +. The input pointer is advanced to point to the next symbol 100.

Stack	**Remaining Input**
$ | E | +	100 – 56 $

The parser now receives the next token ‘100’, which is an ID from the lexical analyser. The parser compares the precedence of ‘+’ (The topmost terminal symbol on the stack) with ID by viewing the entry O[+][ID]. The value is ‘<•’, which implies that the input symbol ID should be pushed on to the stack (shift operation) and the input pointer advanced. The topmost terminal symbol on the stack is ID.

Stack	**Remaining Input**
$ | E | + | ID	– 56 $

The next input is ‘–’, which is a token of type MINUS as classified by the lexical analyser. The parser compares the precedence of ID (which is the topmost terminal symbol of stack) with ‘–’ by viewing the entry O [ID] [–]. The value is ‘•>’, which implies that a reduce operation needs to be carried out. Symbols are popped out until the precedence relation between the top of stack element and the most recently popped element is ‘<•’. The first symbol to be popped out of the stack is ID. This makes the most recently popped out element as ID. The stack then has ‘+’ as the top of the stack. The precedence relation between ‘+’ (topmost terminal symbol on the stack) and the most recently popped out element (ID) is determined by the entry O[+][ID]. The value is ‘<•’, which signals us to stop popping out the elements from stack. The elements that were popped out during this process are ID. We reduce it to a non-terminal E. Note that we are not distinguishing between non-terminals, it is always E even if there were many non-terminals in the operator grammar. We need to push E onto the stack. The topmost ***terminal*** symbol on the stack is however +. Note that the input pointer is not advanced in the reduce operation.

Stack	**Remaining Input**
$ | E | + | E	– 56 $

We continue to use ‘–’ as the next input symbol. The parser compares the precedence of + (topmost terminal symbol on the stack) with ‘–’ with by viewing the entry O[+][–]. The value is ‘•>’, which implies that a reduce operation needs to be carried out. Symbols are popped-out until the precedence relation between the top of stack element and the most recently popped element is ‘<’. The first symbol to be popped out of the stack is non-terminal E. Note that this does not change the most recently popped out terminal, since we had popped out a non-terminal. We pop again, the element to be popped out is ‘+’. This makes the most recently popped out element as ‘+’. The stack then has E as the top of the stack. We pop out again. The top of the stack is now $, which is a non-terminal. The precedence relation between $ and + (the most recently popped out terminal) as determined by O[$][+] is ‘<•’. This signals us to stop popping out the elements from stack. The elements that were popped out during this process were E, + and E in that order. We reduce these elements to a non-terminal E, i.e. E → E + E. Note that we are not distinguishing between non-terminals, it is always E in the case of operator grammar. We need to push E onto the stack.

The topmost terminal symbol on the stack is, however, $. Note that the input pointer is not advanced in the reduce operation.

Stack		Remaining Input
$	E	– 56 $

We continue to use '–' as the next input symbol. The parser compares the precedence of $ (topmost terminal on the stack) with '–' with by viewing the entry O[$][–]. The value is '<•', which implies that the input symbol '–' should be pushed on to the stack (shift operation) and the input pointer advanced. The topmost terminal symbol on the stack is '–'.

Stack			Remaining Input
$	E	–	56 $

The parser now receives the next token '56', which is an ID from the lexical analyser. The parser compares the precedence of '–' (The topmost terminal symbol on the stack) with ID by viewing the entry O[–][ID]. The value is '<•', which implies that the input symbol ID should be pushed on to the stack (shift operation) and the input pointer advanced. The topmost terminal symbol now on the stack is ID.

Stack				Remaining Input
$	E	–	ID	$

The parser now receives the next token '$', the end of input marker from the lexical analyser. The parser compares the precedence of 'ID' (The topmost terminal symbol on the stack) with $ by viewing the entry O[ID][$]. The value is '•>', which implies that a reduce operation needs to be carried out. Symbols are popped out until the precedence relation between the top of stack element and the most recently popped element is '<•'. The first symbol to be popped out of the stack is ID. This makes the most recently popped out element as ID. The stack then has '–' as the top of the stack. The precedence relation between '–' (topmost terminal symbol on the stack) and the most recently popped out element (ID) is determined by the entry O[–][ID]. The value is '<•', which signals us to stop popping out the elements from stack. The elements that were popped out during this process are ID. We reduce it to a non-terminal E, i.e. E → ID. We need to push E onto the stack. The topmost terminal symbol on the stack is, however –. Note that the input pointer is not advanced in the reduce operation.

Stack				Remaining Input
$	E	–	E	$

We continue to use '$' as the next input symbol. The parser compares the precedence of – (topmost terminal symbol on the stack) with '$' with by viewing the entry O[–][$]. The value is '•>', which implies that a reduce operation needs to be carried out. Symbols are popped out until the precedence relation between the top of stack element and the most recently popped element is '<'. The first symbol to be popped out of the stack is non-terminal E. Note that this does not change the most recently popped out terminal, since we had popped out a non-terminal. We pop again, the element to be popped out is '–'. This makes the most recently popped out element as '–'. The stack then has E as the top of the stack. We pop out again. The top of the stack is now $, which is a terminal. The precedence relation between $ and – (the most recently popped out terminal) as determined by O[$][–] is '<•'. This signals us to stop popping out the elements from stack. The elements that were popped out during this process were E, – and E in that order. We reduce these elements to a non-terminal E, i.e. E → E – E. We need to push E onto the stack. The topmost terminal symbol on the stack is, however, \$. Note that the input pointer is not advanced in the reduce operation.

Stack	Remaining Input
\$ | E	\$

We continue to use '\$' as the next input symbol. Now the top most terminal on the stack is '\$' and also the next input symbol is '\$', which signals the completion of a successful parse. This terminates by ***accepting the input string as conformant*** to the grammar. The reader needs to observe how the operator relations table facilitated the shift/reduce decisions made by the parsing program.

Let's now understand how error recovery can be done in operations precedence parsing.

3.5.4.1 Error Reporting and Recovery in Operator Precedence Parsing

In the operator precedence parsing algorithm shown in Algorithm 3.3 there are two points where the operator precedence parser can detect errors.

1. No precedence relation exists between the top of the stack and the current input. For example, considering the grammar in Table 3.18 (which is manifested as a precedence table in Table 3.18), if an erroneous input of say '35 59' is given as an input, the operation precedence parser indicates an unsuccessful parse in step 3 due to the erroneous entry in the precedence table (see below).

Step	Stack	Input	Comment
1	\$	35 59 \$	Initial configuration '\$' is on the top of stack
2		59 \$	ID is shifted on to the stack
3	\$ ID	59 \$	The parser announces an error since O[ID][ID] is an error entry

2. The handle emerging out of the popped elements does not form the RHS of a production. For example, if an erroneous input of say '35 +' is given, the operator precedence parser indicates a unsuccessful parse in step 5 due to the handle not matching the RHS of a production.

Step	Stack	Input	Comment
1	\$	35 + \$	Initial configuration '\$' is on the top of stack
2	\$ ID	+ \$	ID is shifted on to the stack
3	\$ E	+ \$	ID is reduced to E
4	\$ E +	\$	+ is shifted onto the stack
5	\$ E +		The handle resulting out of the popped elements from the stack is 'E +', which does not form the RHS of any production

We can build intelligence in both of these points to indicate not only the error but also the source of error. We can also recover from the current error to proceed with the syntax analysis of the following lines.

In situations where there is no precedence relation existing between the topmost terminal on the stack and the next input, the source of error can be determined by virtue of the position in the operator precedence table. For example, the entry O[ID][ID] is referenced in situations where the input contains two consecutive IDs without an operator between them, as we saw above for the input '35 59'. We can issue a diagnostic message, 'Missing Operand' in such cases. Similarly some other error messages can also be flashed on the position in the relations table. Table 3.20 and Table 3.21 show all the error entries and their corresponding diagnostic messages.

Table 3.20 *Precedence relations table with well-defined error entries*

	+	-	*	/	^	ID	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
–	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
ID	•>	•>	•>	•>	•>	ERR1	ERR1	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	≐	ERR4
)	•>	•>	•>	•>	•>	ERR1	ERR1	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	ERR2	ERR3

The diagnostic messages are as follows:

Table 3.21 *Diagnostics messages*

Error	Diagnostic message
ERR1	Missing operator
ERR2	Missing left parenthesis
ERR3	Missing operand
ERR4	Missing right parenthesis

In all of these situations where we can flash the error messages as indicated by the diagnostics messages in Table 3.21, we can also proceed with the syntax analysis of the next line without stopping at the error by filling in the missing element on the stack. For example, in case of errors resulting in ERR1 message, we can insert an operator on the stack in order to proceed with the syntax analysis of the next line after flashing the error message. Similarly for ERR2, we can insert a left parenthesis in the stack, and allow the syntax analysis to proceed after flashing the ERR2 diagnostic message.

Coming to the situations where the handle popped out does not match the RHS of any production, we could flash a message depending on the resemblance to a particular production. For example, if the handle is 'E +' (as we saw in the earlier example), the production that it resembles is $E \rightarrow E + E$. The missing element is another ID, which would have been reduced to E. We can issue a diagnostic message indicating a 'missing operand' for this case. As a general rule for the grammar in Table 3.18, we can potentially check if there are non-terminals on either end of other operators like –,^,* or /, if they do not exist, then we can issue a diagnostic message 'missing operand'. Additionally, as error recovery, we can also insert the E on the stack and proceed with the syntax analysis of the next line. Similarly, from the characteristics exhibited by the grammar in Table 3.18, we can deduce that a non-terminal needs to be present between the parentheses or else we can flash a diagnostic message indicating that the expression is missing between the parentheses. The table below shows some of the checks that can be made for the grammar in Table 3.18 with regard to the handle popped out during a reduction.

Check	Diagnostic message on failing the check	Related production
Operators +.–,*,/,^ should be surrounded on either side by non-terminal E	Missing operand	$E \rightarrow E + E$ $E \rightarrow E - E$ $E \rightarrow E * E$ $E \rightarrow E / E$ $E \rightarrow E \wedge E$
There should be a non-terminal between open parenthesis, and close parenthesis	No expression between parentheses	$E \rightarrow (E)$

Both of the above-mentioned methods of error recovery involve some kind of local correction either in the stack or input or both. Thus the error recovery schemes in operator precedence parsing fall under the category of phrase level recovery as described in Section 3.3.

3.5.4.2 *Precedence Functions*

The main use of the operator precedence table is for determining the precedence of a terminal with respect to another. For operator grammar with many rules, it could become large. Alternatively, it is also possible to define two functions ***f*** and ***g*** such that

$f(a) < g(b)$ whenever $a < b$

$f(a) = g(b)$ whenever $a = b$

$f(a) > g(b)$ whenever $a > b$

where 'a' and 'b' are the terminals for whom we need to determine the precedence relations.

These functions ***f*** and ***g*** are called as ***precedence functions***. The precedence functions help in reducing the memory consumption of the operator precedence parser by virtue of eliminating the table and replacing with the functions. The precedence functions equivalent of the operator precedence relations table in Table 3.19 is shown in Table 3.22.

Table 3.22 *Precedence functions*

	+	–	*	/	^	(	)	ID	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

In the parsing algorithm, we need to compute f(a) and g(b) instead of looking at the table entry O[a][b]. However, one disadvantage with the precedence functions is that we lose the ability to detect errors based on the error entries of the table as explained in the previous section.

3.5.4.3 *Advantages and Disadvantages of Operator Precedence Parsing*

The following are the advantages and the disadvantages of operator precedence parsing:

Advantages

- It is a simple and easy to implement parsing technique.
- The operator precedence parser is constructed by hand after understanding the grammar. It is simpler to debug.

Disadvantages

- It is hard to handle tokens like minus (–), which has two different values of precedence depending on whether it is being used as binary or unary operator.
- This technique unlike most of the other techniques we have seen does not take the grammar as the input and generate a parser. The parser has a fragile relationship with the grammar. Any addition or deletion of production rules would require a rewrite of the parser. Because of the implicit nature of the dependency on the grammar rules (for example, in the error recovery), it might so happen that the parser might not accept sentences belonging to the language or reject the sentences belonging to the language.
- The operator precedence parsing technique can parse only a small class of grammars (operator grammars).

In the Section 3.5.4.4, we see a functional operator precedence parser with error detection facility.

3.5.4.4 *Example 7—An Operator Precedence Parsing Program* This section shows an operator precedence parsing program built for the grammar shown in Table 3.18. Algorithm 3.3 is used in this example. The program is capable of performing error reporting. The following dialog shows how the operator precedence parsing program can be used to perform syntax analysis on given expressions.

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oop-prec-lex.c op-prec-lex.l

# Compiling the Lexical Analyzer
$ gcc -c -o op-prec-lex.o op-prec-lex.c

# Building ex7 Binary
$ g++ -g -Wall ex7.cc op-prec.cc op-prec-lex.o -o ex7

# Simple Expression with 2 numbers
$ ./ex7 '25 + 35'
Shifting ID [25]
Reducing E --> ID
Shifting + [+]
Shifting ID [35]
Reducing E --> ID
Reducing E --> E + E
25 + 35
SYNTAX CORRECT

# Simple Expression with a number and a variable
$ ./ex7 'a + 35'
Shifting ID [a]
Reducing E --> ID
Shifting + [+]
Shifting ID [35]
Reducing E --> ID
Reducing E --> E + E
a + 35
SYNTAX CORRECT
```

```
# Expression with multiple Operators
$ ./ex7 'a + 35 * 40'
Shifting ID [a]
Reducing E --> ID
Shifting + [+]
Shifting ID [35]
Reducing E --> ID
Shifting * [*]
Shifting ID [40]
Reducing E --> ID
Reducing E --> E * E
Reducing E --> E + E
a + 35 * 40
SYNTAX CORRECT

# Expression with multiple Operators
$ ./ex7 'a ^ 35 - 40'
Shifting ID [a]
Reducing E --> ID
Shifting ^ [^]
Shifting ID [35]
Reducing E --> ID
Reducing E --> E ^ E
Shifting - [-]
Shifting ID [40]
Reducing E --> ID
Reducing E --> E - E
a ^ 35 - 40
SYNTAX CORRECT

# Expression with missing Operand
$ ./ex7 'a ^ -'
Shifting ID [a]
Reducing E --> ID
Shifting ^ [^]
Missing Operand

SYNTAX INCORRECT

# Expression with missing Operator
$ ./ex7 'a h'
Shifting ID [a]
Missing Operator

SYNTAX INCORRECT
```

3.5.5 LR Parsing

We saw that in operator precedence parsing, the precedence relations table served as a mechanism to make shift/reduce decisions. In LR parsing method a parsing table (also called ***LR parsing table***) is used to detect correct handles and make informed shift/reduce decisions.

The parsing table is used to figure out whether a shift or a reduce needs to be done on the receipt of a given input. In case of a reduce operation, it tells us which production needs to be used for it. Given the current ***'state'*** of the LR parser and the next input symbol, the parsing table helps in arriving at a shift/reduce decision. The starting state of the LR parser is 0. We had earlier seen in Section 3.5.2 that a shift involves pushing the input symbol on the stack, a reduce operation involves popping of as many elements as the right-hand side of the production and a push of the left-hand side of the production.

For the grammar in Table 3.17 (reproduced below), the parsing table is shown in Table 3.23.

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression	→	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR CONSTANT
5		|	c_expression OPERATOR IDENTIFIER

In the parsing table shown in Table 3.23, the current state is given in the first column. The entries sX represent a shift operation and changing the state to X. For example, in state 0, an input of 'IDENTIFIER' involves a shift of the input and a change of the state to 1. The entries with rX denote reduction by production X. For example, in state 4, an input of 'OPERATOR' would mean a reduction by Production 2 (c_expression → CONSTANT).

This parsing table is the key data structure for LR parsing. ***The use of the parsing table to make shift/reduce decisions is the central idea in the LR parsing method***. The generation of the parsing table from the grammar will be the topics for later sections, for now assume the availability of the parsing table to simplify your understanding.

A state diagram can visually represent a parsing table. The state diagram shown in Fig. 3.9 is representative of the parsing table in Table 3.23.

Table 3.23 *Parsing table for an LR parser*

CURRENT STATE (Top of stack)	action						goto	
	IDENTIFIER	CONSTANT	OPERATOR	EQ_ TO_OP	SEMI_ COLON	End of Input ($)	c_expression	c_statement
0	s1	-	-	-	-	-	-	10
1	-	-	-	s2	-	-	-	-
2	s3	s4	-	-	-	-	5	-
3	-	-	r3	-	r3	-	-	-
4	-	-	r2	-	r2	-	-	-
5	-	-	s6	-	s7	-	-	-
6	s8	s9	-	-	-	-	-	-
7	-	-	-	-	-	r1	-	-
8	-	-	r5	-	r5	-	-	-
9	-	-	r5	-	r5	-	-	-
10	-	-	-	-	-	accept	-	-

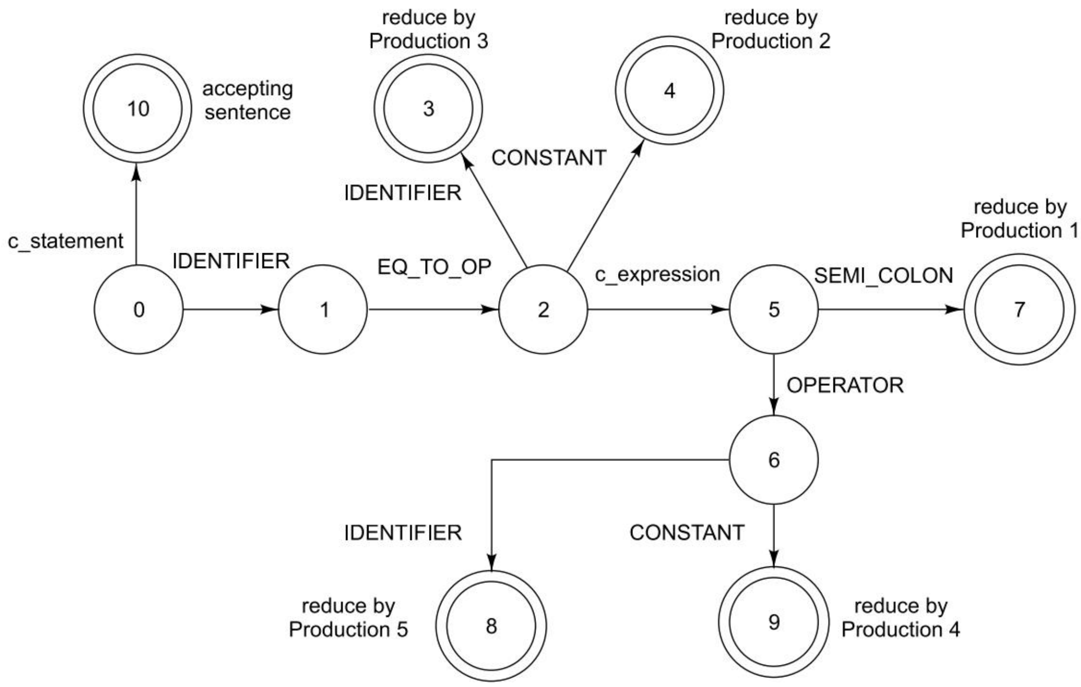


Fig. 3.9 *State machine for c-statement grammar*

We can see from the state diagram that a particular input symbol causes a transition from one state to another. In non-accepting states, the next input symbol causes a shift and a transition to another state, for example, in the state 1, an input of EQ_TO_OP causes a shift operation and a transition to state 2. In accepting states, the next input symbol can cause a shift or a reduce operation, for example, in state 3, an input of SEMI_COLON would trigger a reduce operation using production 3 (c_expression → IDENTIFIER). Even though, in this particular state diagram none of the accepting states transition to another state by shift operation on the receipt of next symbol, It is common to have transitions by shift operation in accepting states too.

An LR parser consists of an input, an output, a stack, driver program and a parsing table made up two parts (action and goto). Figure 3.10 shows the various components of an LR parser.

The stack consists of states. These are the same states that we had earlier seen in the state diagram of Fig. 3.9. The top of the stack and the next symbol are used to index into a parsing table to make a shift-reduce decision.

The parsing table consists of two parts, a parsing action part called the ***action*** and a goto function called as ***goto***. The action table is a two-dimensional array indexed by state and the next input symbol, i.e. action[state][input]. An action table entry can have one of following four kinds of values in it:

1. shift X, where X is a state number
2. reduce X, where X is a production number
3. accept, signifying the complete of a successful parse
4. error entry

The goto table is a two-dimensional array indexed by state number and a non-terminal grammar symbol, i.e. goto[state][non-terminal]. A goto entry contains a state number. Both the action table and goto table are derived from the grammar of the language. Table 3.23 shows the parsing table for the grammar in Table 3.17.

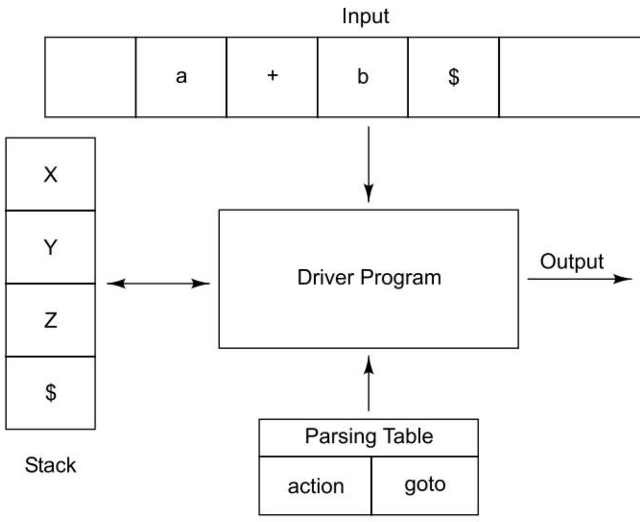


Fig. 3.10 *LR parser components*

The driver program uses the current state C (given by top of stack element), the next input symbol 'a' to consult the entry at action[C][a]. The driver program makes one of the four actions as dictated by the content of the entry in the action table.

1. If action[C][a] = shift X, the parser executes a shift of X on to the top of stack and advances the input pointer.
2. If action[S][a] = reduce X, the parser executes a reduce using the production X. The reduce operation involves popping of as many number of elements as existing on right-hand side of production X. The driver program then pushes the left-hand side of the production. The input pointer is not advanced in this operation.
3. If action[S][a] = accept, then parsing is complete, and the sentence is accepted.
4. If action[S][a] = error, then the parser has discovered an error and calls error recovery routine.

The algorithm followed by the driver program is shown in Algorithm 3.4.

```
action and goto are 2 dimensional arrays
TOS is used to denote Top Of Stack
```

```
push 0
while ( action[TOS][input] != accept)
{
        if( action[TOS][input] == sX){
              push (X);
              advance();
        } else if (action[TOS][input] == rX){
              pop ( the no of elements in the RHS of production X) ;
              push ( goto [new TOS][LHS of production X]
        } else {
              return (FAILURE)
        }
}
return (SUCCESS)
```

Algorithm 3.4 *LR parsing*

A step-by-step illustration of LR parsing of an input will make Algorithm 3.4 and the concepts clear. Consider an LR parser working on an input 'c = a +1;' using Table 3.23 as its parsing table. Initially in LR parsing, the start state (state 0) is pushed on to the stack.

Stack	**Remaining Input**
$\boxed{0}$	c = a + 1 ; $

The parser receives the first token 'c' which is an IDENTIFIER from the lexical analyser. The action[0][IDENTIFIER] = s1, which implies that we push state 1 onto to the stack and advance the input pointer.

Stack	**Remaining Input**
$\boxed{0}\boxed{1}$	= a + 1 ; $

The next input is '=', which is a token of type 'EQ_TO_OP' as classified by the lexical analyser. The action[1][EQ_TO_OP] = s2, which implies that we push state 2 on to the stack and advance the input.

Stack	**Remaining Input**
$\boxed{0}\boxed{1}\boxed{2}$	a + 1 ; $

The next input is 'a', which is a token of type IDENTIFIER as classified by the lexical analyser. The action[2][IDENTIFIER]= s3, which implies that we push state 3 on to the stack and advance the input pointer. Note that the state machine has avoided the reducing the IDENTIFIER into C expression. The state table is clearly identifying the 'right-handle' in the parse process.

Stack	**Remaining Input**
$\boxed{0}\boxed{1}\boxed{2}\boxed{3}$	+ 1 ; $

The next input token is '+', which is a token of the type OPERATOR as classified by lexical analyser. From the action table, action[3][OPERATOR]= r3, which implies that we reduce using the rule 3. Rule 3 is 'c_expression → IDENTIFIER'. The number of elements on the right-hand side of the production is 1. Hence we pop one element from the top of stack.

Stack	**Remaining Input**
$\boxed{0}\boxed{1}\boxed{2}$	+ 1 ; $

The new top of stack is state 2. The left-hand side of the production 3, that we are reducing is c_expression. Now, the parser looks at goto[2][c_expression], which is state 5. The parser pushes state 5 on to the stack. Note that input pointer is not advanced during the reduction.

Stack	**Remaining Input**
$\boxed{0}\boxed{1}\boxed{2}\boxed{5}$	+ 1 ; $

We continue using the input token '+' of the type OPERATOR, since the last reduce did not advance the input pointer. From the action table, action[5][OPERATOR]= s6, this implies that we push state 6 on the stack and advance the input pointer.

Stack	**Remaining Input**
$\boxed{0}\boxed{1}\boxed{2}\boxed{5}\boxed{6}$	1 ; $

The next input token is ‘1’, which is of the type ‘CONSTANT’ as classified by the lexical analyser. From the action table, action[6][CONSTANT]= s9, which implies that we shift state 9 on to the stack and advance the input pointer.

Stack **Remaining Input**

0	1	2	5	6	9

; $

The next input token is ‘;’, which is of the type SEMI_COLON as classified by the lexical analyser. From the action table, action[9][SEMI_COLON] = r5, which implies that we reduce using the rule 5. Rule 5 is c_statement → c_expression OPERATOR IDENTIFIER. The number of elements on the right-hand side of the production is 3. Hence we pop three elements from the top of stack.

Stack **Remaining Input**

0	1	2

; $

The new top of stack is state 2. The left-hand side of the rule 4 that we are reducing is ‘c_statement’. Now, the parser looks at goto[2][c_statement], which is state 5. The parser pushes state 5 on to the stack. Note that input pointer is not advanced during the reduction.

Stack **Remaining Input**

0	1	2	5

; $

We continue using the input token ‘;’ of the type SEMI_COLON, since the last reduce did not advance the input pointer. From the action table, action[5][SEMI_COLON]= s7, this implies that we push state 7 on the stack and advance the input pointer.

Stack **Remaining Input**

0	1	2	5	7

$

The next input is ‘$’, which is indicative of the end of the input. From the action table, the action[7][‘$’] = r1, which implies that we reduce using the rule 1. Rule 1 is c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON. The number of elements on the right-hand side of the production is 4. Hence we pop four elements from the top of stack.

Stack **Remaining Input**

0

; $

The new top of stack is state 0. The left-hand side of the rule 1 that we are reducing is ‘c_statement’. Now, the parser looks at goto[0][c_statement], which is state 10. The parser pushes state 10 on to the stack. Note that input pointer is not advanced during the reduction.

Stack **Remaining Input**

0	10

$

We continue using the input token ‘$’ which is the end-of-input, since the last reduce did not advance the input pointer. From the action table, action[10][$]= accept, this implies that the input string is ***accepted*** by the grammar of the language.

In the next section we take a look at an LR parser program that implements Algorithm 3.4.

3.5.5.1 *Example 8—An LR Parser* In this section, we shall take a look at an LR parser program that implements Algorithm 3.4. The idea here is to use an already constructed parsing table (i.e.) the action and goto tables shown in Table 3.23. Remember that the parsing table in Table 3.23 is a manifestation of the grammar in Table 3.17. The method for constructing the parsing table from the grammar will follow in later sections. The focus right now is on the algorithm that performs the syntax analysis on the input given a parsing table.

The following dialog shows how to build the binary and use it to verify the syntax of input.

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the Lexical Analyzer
$ gcc -c -o c-stmt-lex.o c-stmt-lex.c

# Building ex8 Binary
$ g++ -g -Wall -DCHAP3_EX8 ex8.cc c-stmt-lr-parse.cc c-stmt-lex.o -o ex8

# Variant 1
$ ./ex8 'count=5;'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 258 (CONSTANT)
Shifting token 258 (CONSTANT), Entering state 2
Reading a token: Next token is 257 (EQ_TO_OP)
Shifting token 257 (EQ_TO_OP), Entering state 4
Reading a token: Next token is 256 (SEMI_COLON)
Reducing via Rule 2
Entering state 5
Shifting token 256 (SEMI_COLON), Entering state 7
Reading a token: Now at end of input.
Reducing via Rule 1
Entering state 10
Success
count=5;
SYNTAX CORRECT

# Variant 5
$ ./ex8 'count=count+1;'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 258 (CONSTANT)
Shifting token 258 (CONSTANT), Entering state 2
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 3
Reading a token: Next token is 260 (OPERATOR)
Reducing via Rule 3
Entering state 5
```

```
Shifting token 260 (OPERATOR), Entering state 6
Reading a token: Next token is 257 (EQ_TO_OP)
Shifting token 257 (EQ_TO_OP), Entering state 9
Reading a token: Next token is 256 (SEMI_COLON)
Reducing via Rule 4
Entering state 5
Shifting token 256 (SEMI_COLON), Entering state 7
Reading a token: Now at end of input.
Reducing via Rule 1
Entering state 10
Success
count=count+1;
SYNTAX CORRECT

# Missing Identifier / Constant
$ ./ex8 'count=5+;'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 258 (CONSTANT)
Shifting token 258 (CONSTANT), Entering state 2
Reading a token: Next token is 257 (EQ_TO_OP)
Shifting token 257 (EQ_TO_OP), Entering state 4
Reading a token: Next token is 260 (OPERATOR)
Reducing via Rule 2
Entering state 5
Shifting token 260 (OPERATOR), Entering state 6
Reading a token: Next token is 256 (SEMI_COLON)
Error not finding entry action_table[6][256]..Exiting
count=5+;
        ^
SYNTAX INCORRECT
```

3.5.5.2 *Construction of LR Parsing Table* In the last section, we had focused on parsing an input given the parsing table (action and goto tables). We understood that the parsing table is derived from the grammar of the language. The LR parsing program described previously depends on the parsing table for performing the syntax analysis of the input. Note that the algorithm does not undergo any change while catering to different grammars, only the parsing table changes. In this section, we shall see how to construct the parsing table from a given grammar.

There are 3 major methods for constructing the parsing table from the grammar. They are:

1. Simple LR or SLR(1) method.
2. Canonical LR or LR(1) method.
3. Look ahead LR or LALR(1) method.

The simple LR method of constructing the parsing table uses look-ahead information of 1 input symbol. The way it uses the look-ahead information to construct entries for the parsing table differs from LR(1) and LALR(1). This method succeeds in constructing the parsing table for a sub-set of LR grammar called SLR(1) grammar.

The canonical LR method of constructing parsing table entries is the most powerful method compared to all other methods. This succeeds in constructing a parsing table for a large class of grammars called as LR(1) grammars. However, the disadvantage is that it generates a large parse table. An optimisation in terms of size of parsing table is the basis of LALR method.

The look-ahead LR method is widely used in practice. This method optimises the size of parsing table generated by canonical LR method, by collapsing a few states into a single state. Due to the collapsing of states, sometimes loss of information happens. Hence, it succeeds in constructing a parsing table for a subset of LR(1) grammar called as LALR(1) grammar. However, LALR(1) grammar is sufficient to express most of the grammars of programming languages. The most commonly used parser generators like bison, yacc, etc. employ this method for constructing the parsing table.

All these methods use the 3 steps shown in Fig. 3.11 for constructing the parsing table from the grammar. In the first step, an extra production rule is added to the original set of productions to create an augmented grammar G'. In the second step, we create a canonical collection of sets of entities called as Items using two distinct functions called the goto and closure functions operating on the augmented grammar G'. In step three, we convert the canonical collection of sets of Items into the parsing table by applying certain rules. We will deal with details of each of these steps with respect to all of the methods a little later.

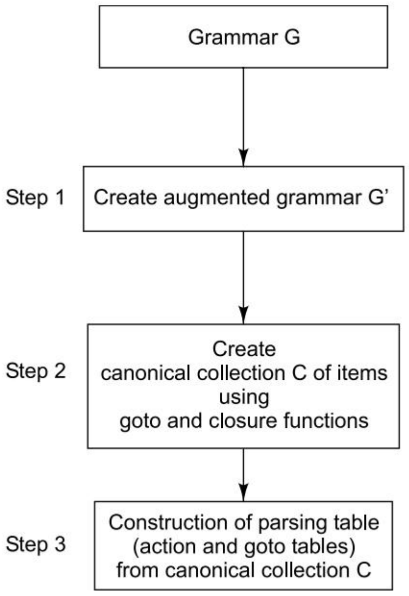

Grammar G
Step 1
Create augmented grammar G'
Step 2
Create canonical collection C of items using goto and closure functions
Step 3
Construction of parsing table (action and goto tables) from canonical collection C

Fig. 3.11 *Constructing an LR parsing table from the grammar G*

In the first step of creating augmented grammar from the grammar, all the 3 methods—SLR, canonical LR and LALR follow the same procedure.

In the second step of creating the canonical collection C of entities called items, the SLR uses an entity known as LR0 item, while canonical LR and LALR use a more specialised entity called as LR1 Item. The goto and closure functions that aid in the creation of canonical collection C are common for both canonical LR and LALR methods. The SLR method has a different goto and closure functions, when compared to the respective counterparts in canonical LR and LALR methods.

In the final step of constructing the parsing table from the canonical collection, each of the three methods apply a different mechanism. The SLR does consider the look-ahead information, but uses it in a primitive fashion to construct the parsing table from the collection of items. The canonical LR applies the best criteria to construct the parsing table entries from the collection of item sets. The LALR is a minor variant of canonical LR approach, which focuses on optimising space rather than adopting a different method to construct the parsing table entries.

The SLR(1) method is described in Section 3.5.5.3, the LR(1) method in Section 3.5.5.4 and LALR(1) method in Section 3.5.5.5.

3.5.5.3 Construction of Parsing Table by SLR(1) method This section talks about the SLR(1) method for constructing a parsing table. This parsing table construction method is the simplest of all the methods and succeeds for SLR(1) grammars.

Before we discuss in detail about the SLR(1) method of constructing the LR parse table, let's understand a few definitions.

Important Definitions This section discusses important definitions that would be used in algorithms to construct the parsing table.

Augmented Grammar If G is a grammar with start symbol S then the ***augmented grammar*** G′ for G consists of all the productions in G and an additional production S′ → S. The start symbol for G′ is S′.

Consider a grammar G, consisting of following productions, where c_statement is the start symbol.

1	c_statement	→	IDENTIFIER	EQ_TO_OP	c_expression	SEMI_COLON
2	c_expression	→	CONSTANT			
3		|	IDENTIFIER			
4		|	c_expression	OPERATOR	CONSTANT	
5		|	c_expression	OPERATOR	IDENTIFIER	

The augmented grammar for G is:

Table 3.24 *Augmented grammar for the grammar in Table 3.17*

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression	→	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR CONSTANT
5		|	c_expression OPERATOR IDENTIFIER
6	**c_statement′**	→	**c_statement**

The main use of augmented grammar lies in the fact that the acceptance of input happens, when the parser is about to reduce production S′ → S. The use of the augmented grammar can be appreciated better in situations where the start symbol has multiple productions where it is the left-hand side of the production. It would be difficult to keep track of all the productions with the start symbol as the LHS and find out if the acceptance of input has happened. But, with the usage of this extra production, the state machine needs to keep track of only one state where the reduction of S′ → S is envisaged. This state would trigger the acceptance of the input.

In the above example, the acceptance of the input string happens when we are about to reduce the production c_statement′ → c_statement.

We shall see how the augmented grammar comes in handy for computing the closure and goto operations discussed next.

LR(0) Item An ***LR(0) Item*** (or simply ***Item*** for short) of a grammar G is a production of G with a dot in some position of the right-hand side. For a grammar consisting of a production X → ABC, the LR(0) Items are:

X → •ABC

X → A•BC

X → AB•C

X → ABC•

For an epsilon production X → ε there is only one LR(0) Item denoted by

X → •

An LR(0) Item can be represented by a 2-tuple of (production, position of dot).

The dot can be used for indicating two aspects:

- To show the portion of input that is already consumed. In the case of shift-reduce parsing, it can be used to indicate what is already present on the stack. If the dot moves to the rightmost point of the production, we can use it to 'reduce' by that production.
- The symbols on the right of the dot can be used as look ahead symbols.

These LR(0) Items form the basis of states in the state diagram (and the parsing table), an example of which is shown in Fig. 3.9. An LR(0) Item is associated with one of the states in the state diagram. The same item cannot be in multiple states. Figure 3.12 shows a few Items associated with some of the states for the grammar in Table 3.17.

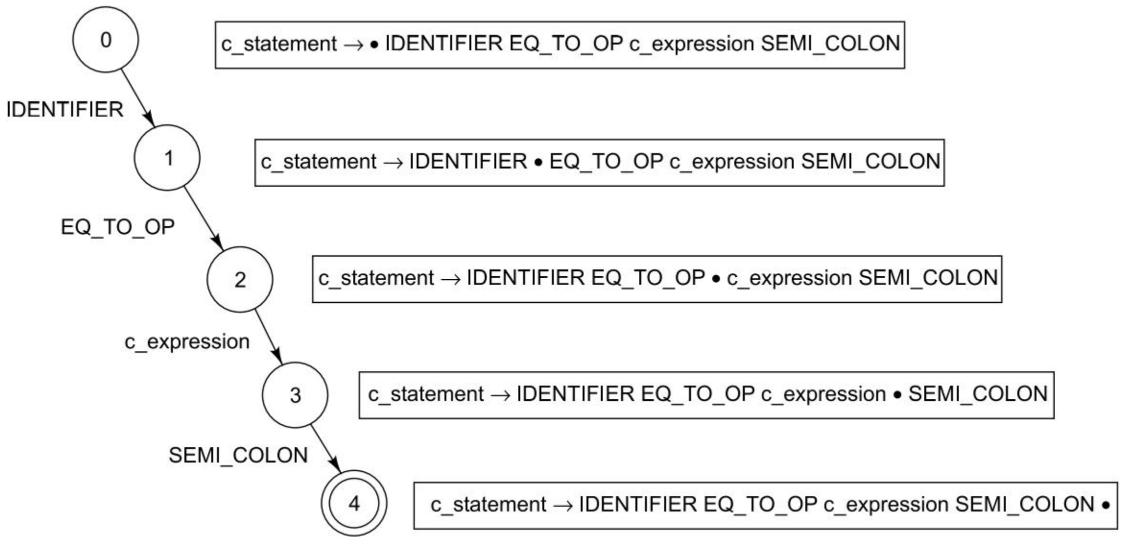

0
c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
IDENTIFIER
1
c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON
EQ_TO_OP
2
c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON
c_expression
3
c_statement → IDENTIFIER EQ_TO_OP c_expression • SEMI_COLON
SEMI_COLON
4
c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON •

Fig. 3.12 *LR(0) Items associated with states*

An Item related to Production 1 accompanies each of the states shown in the Fig. 3.12. For the item in state 0, the dot is before the IDENTIFIER. This tells us that the parser is expecting an IDENTIFIER as the next input symbol. In state 1, the dot has shifted by one position, now it is before EQ_TO_OP. This tells us that the parser has already consumed an IDENTIFIER (shifted it on to the stack) and is now expecting an EQ_TO_OP symbol. In state 4, the dot has reached the extreme right position. This tells us that the entire right-hand side of the production is on the stack, and hence we can reduce by Production 1 (and hence the concentric circles for representing the state 4). Note that Fig. 3.12 consciously shows the items related to Production 1 only in order to illustrate the concept better. In reality there are items related to other productions also in any given state.

Closure Operation Now, consider the state 2 of Fig. 3.12, where we are expecting a c_expression (non-terminal) as determined by the item c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON. This means that we are expecting an input symbol depending on what the productions with 'c_expression' as the left-hand side indicate. The following productions qualify.

2	c_expression	→	CONSTANT		
3	c_expression	→	IDENTIFIER		
4	c_expression	→	c_expression	OPERATOR	CONSTANT
5	c_expression	→	c_expression	OPERATOR	IDENTIFIER

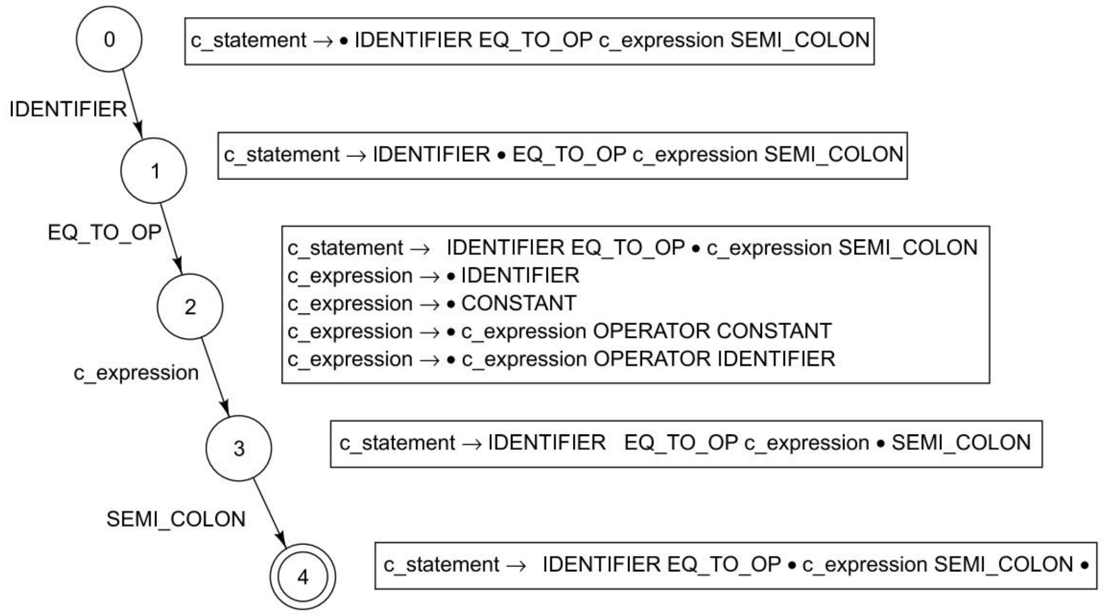
Looking at the productions, we can deduce that, in state 2, we are really expecting inputs as determined by the following items:

c_statement	→		IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON			*The original one*
c_expression	→	•	CONSTANT			*The derived ones*
c_expression	→	•	IDENTIFIER			
c_expression	→	•	c_expression	OPERATOR	CONSTANT	
c_expression	→	•	c_expression	OPERATOR	IDENTIFIER	

This set of items is called as the ***closure set*** of item c_statement → IDENTIFIER EQ_TO_OP_• c_expression SEMI_COLON, the operation by which it was derived is called as ***closure operation.***

Adding the above items to state 2, we have Fig. 3.13. This brings home the concept that items grouped together form a state. As we saw, the item 'c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON' was the seed for the other items in state 2 begotten by the closure operation. Note that, all the additional items generated out of closure operation have the dot at the left end of RHS, while the original item does not have a dot at the left end.

Those items that do not contain a dot at the far left are called as the ***seed items*** or ***kernel items***. These kernel items form the basis of forming the other items of the state. The other items that contain a dot at the left end are called as the ***non-kernel or non-seed items***. The kernel item in state 2 is c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON, while the other items in state 2 are the non-kernel items. There is one exception to this rule, which is the Item S′ → • S, where S is the start symbol of grammar G and S′ is the start symbol of the augmented grammar G′. Even though it has a dot at the far left of the RHS, it is called as a seed item, because it is the one that seeds the start state of the state diagram.



0
c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
IDENTIFIER
1
c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON
EQ_TO_OP
2
c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON
c_expression → • IDENTIFIER
c_expression → • CONSTANT
c_expression → • c_expression OPERATOR CONSTANT
c_expression → • c_expression OPERATOR IDENTIFIER
c_expression
3
c_statement → IDENTIFIER EQ_TO_OP c_expression • SEMI_COLON
SEMI_COLON
4
c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON •

Fig. 3.13 *More LR(0) Items associated with states*

Let's get to a more formal definition for the computation of closure operation. If I is a set of items for a grammar G, then closure of I, represented by closure(I) can be computed using the following 2 rules.

Table 3.25 *Rules for computing closure of an LR(0) item set*

Rule 1	Every element of I is added to closure of I
Rule 2	If X → A•BC, is in closure(I), and there exists a production B → b_1 b_2 .. b_n, then add item B → •b_1 b_2 .. b_n if it is not already in closure(I). Keep applying this rule until there are no more elements added.

Let's check out an example to understand the nuances of closure set computation. Consider a grammar with the following productions:

1	my_start_sym	→	my_non_term_a MY_TERM_ONE
2	my_non_term_a	→	my_non_term_b MY_TERM_TWO
3		|	MY_TERM_THREE
4	my_non_term_b	→	MY_TERM_FOUR my_non_term_c
5	my_non_term_c	→	MY_TERM_FIVE

Let's compute the closure of the item set I, where

I = my_start_sym → • my_non_term_a MY_TERM_ONE

Using rule 1, the closure set of I would have the item my_start_sym → • my_non_term_a MY_TERM_ONE. By rule 2, we get other element my_non_term_a → • my_non_term_b MY_TERM_TWO

We have the two elements in the first pass

Closure(I) = my_start_sym → • my_non_term_a, MY_TERM_ONE
my_non_term_a → • my_non_term_b MY_TERM_TWO

We apply rule 2 again on the above set since the rule 2 says that we keep applying it until there are no new elements added. We need to add the item whose left-hand side is my_non_term_b and the dot is at the start of the RHS. Thus, we have now 3 elements.

Closure(I) = my_start_sym → • my_non_term_a, MY_TERM_ONE
my_non_term_a → • my_non_term_b MY_TERM_TWO
my_non_term_b → • MY_TERM_FOUR my_non_term_c

Applying rule 2 again does not yield any new item. Hence we stop with this pass. The closure set is

Closure(I) = my_start_sym → • my_non_term_a, MY_TERM_ONE
my_non_term_a → • my_non_term_b MY_TERM_TWO
my_non_term_b → • MY_TERM_FOUR my_non_term_c

The algorithm for computing closure(I) is formalised in Algorithm 3.5.

```
I is a set of items belonging to grammar G
At the end of the algorithm J contains the closure(I)
```

```
J=I
do{
        added=0
        for ( each item X → A·BC in J) {
                if ( production B → $b_1b_2b_3..b_n$ exists in grammar G){
                        if ( item B → ·$b_1b_2b_3..b_n$ does not exist in C){
                                add the item B → ·$b_1b_2b_3..b_n$
                                added ++
                        }
                }
        }
} while ( added > 0)
```

Algorithm 3.5 *Closure set computation*

Goto Operation Let's have a preview on how the items for each of the states are computed. The start state is always seeded with an item related to the extra production that we created for the augmented grammar. The beauty of this extra production is that it allows us embrace all the productions with the start symbol as the left-hand side by means of closure operation.

Let us consider the grammar of Table 3.17 whose start symbol is the c_statement. As we saw earlier, the augmented grammar had an extra production *c_statement′→ c_statement*. The start state is seeded with the item

I_0 = {c_statement′ → • c_statement}

Applying closure on the above set, would get us the items with all the productions that have the start symbols as the left-hand side, namely,

c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON

Applying closure does not yield any extra items since the symbol to the right of the dot is a terminal. Thus, there are two items in the item set for the start state:

I_0 = c_statement′ → • c_statement
c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON

Now, if we input any one of the symbols in the grammar, we need to see what items are generated. This is facilitated by the ***goto*** function.

If I is a set of LR(0) Items which contains an item $A \rightarrow \alpha \bullet X\beta$ where X is grammar symbol, then ***goto(I,X)*** is defined as the closure of the set of all the items $A \rightarrow \alpha X \bullet \beta$. The goto operation gives the set of all the possible items given X as the next input.

Coming back to the example, from the start state item set I_0, let's see what the goto function does.

goto(I_0, IDENTIFIER) = closure set of {c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON}

= {c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON}

goto(I_0, CONSTANT) does not yield any items because there is no item in I_0 that has a dot before CONSTANT. Similarly, there are no items yielding for goto(I_0, OPERATOR), etc.

Thus I_1 = {c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON} can be used to represent state 1.

The goto(I,X) is formalised in Algorithm 3.6.

```
I, is the set of Items for which the goto(I,X) needs to be computed
At the end of the Algorithm J contains goto(I,X)

for ( each item A → α·Xβ in I)
        Add the item A → αX·β to tmp_set
}
J= closure(tmp_set)
```

Algorithm 3.6 *goto(I,X) for a set of LR(0) items*

We have just seen how goto function helps us in creating items for other states from a given state. Now, armed with the concepts of augmented grammar, LR(0) item, closure and goto functions, lets get to the procedure for constructing a parsing table using SLR(1) Method.

Procedure for constructing the parsing table Let's revisit the steps shown in Fig. 3.11 with reference to SLR(1) method of constructing the parsing table.

In step 1, we construct the augmented grammar (G′) by adding an additional production $S' \rightarrow$ S where S is the start symbol of the original grammar (G).

Step 2 involves creation of a canonical collection of sets of LR(0) items for SLR(1) parsing table. We start with the Item set 0 (I_0) containing merely the additional production of augmented grammar and the closure obtained on it. This is the first set added in the canonical collection of sets C. Now, for every set I_n present in canonical collection C, we determine goto(I_n, X), for every symbol X in the grammar, if that yields a set not already present in the canonical collection of sets, we add it to the collection. If no sets are added to the collection during one traversal of the entire canonical collection, we halt the algorithm completing the canonical collection of sets of items. This procedure is shown in Algorithm 3.7.

```
I,J, tmp_set are sets of LR(0) items
C is canonical Collection of sets of LR(0) Items
```

```
I= {S'→ · S}
J = closure (I)
Add J as one set in canonical collection C /* This corresponds to state 0 */
do
{
        added = false;
        for ( each set I in canonical collection C)
             for ( each grammar symbol X in symbol table)
                  tmp_set = goto(I,X)
                  if( tmp_set is not present in canonical collection C){
                       added = true;
                       add tmp_set in canonical collection C
                  }
             }
        }
} while ( added == true)
```

Algorithm 3.7 *Creating canonical collection of sets of LR(0) items*

In step 3, we construct the action and goto table using the canonical collection of sets of LR(0) items by using algorithm 3.8. The initial state is constructed from the set containing the element [S′ → S], which is I_0. The following rules form the basis of construction of SLR(1) parsing table from the canonical collection of sets of items.

Table 3.26 *Rules for constructing parsing table from canonical collection*

Rule 1	If there is an item A → $\alpha \bullet X\beta$ in I_i and goto(I_i, X) is in the Item set I_j then action[I][X] = shift j, where X is a terminal
Rule 2	If there is an Item A → $\alpha \bullet$ in I_i then set action[i][X] = reduce by A → α for all terminals X in the FOLLOW(A).
Rule 3	If there is an item S′ → S • in I_i then set action[I][$] = accept
Rule 4	If the goto(I_i, X) = I_j then goto[i][X] = j, where X is a non-terminal.

All the entries not defined by the above rules are error entries. If there is conflict in the entries generated out of the above rules, then the grammar is not SLR(1) grammar. The algorithm fails to produce a parser. These rules are formalised in Algorithm 3.8.

C is canonical collection of sets of Items = {I_0, I_1, I_2,... I_n} created using Algorithm 3.7

```
for ( each item set Ii in Canonical Collection) {
        for ( each item P in a Item set $I_i$) {
                if ( P is S′ → S·){
                        action[I][$]=ACCEPT;
                } else if ( P is of the form A → $\alpha$·){
                        for ( each terminal k in FOLLOW(A)){
                                action[i][k] = Reduce by the Production A → $\alpha$
                        }
                } else if ( P is of the form A → $\alpha$·X$\beta$ and X is a Terminal){
                        if ( goto($I_i$,X) == $I_j$){
                                action[I][X]=j ;
                        }
                } else if ( P is of the form A → $\alpha$·X$\beta$ and X is a NonTerminal) {
                        if ( goto($I_i$,Y) == $I_j$){
                                goto[I][Y]=j ;
                        }
                }
        }
}
```

Algorithm 3.8 *Construction of SLR(1) parsing table from canonical collection of LR(0) items*

An Illustration of SLR(1) Method for constructing a parsing table In this section, we take the grammar shown in Table 3.17 (reproduced below), go over each of the 3 steps and construct a parsing table for the same using the SLR(1) method.

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON		
2	c_expression	→	CONSTANT		
3		|	IDENTIFIER		
4		|	c_expression	OPERATOR	CONSTANT
5		|	c_expression	OPERATOR	IDENTIFIER

In step 1, we create augmented grammar G’ by adding an extra production S’→ S, where S is the start symbol of the original grammar G. So, we have augmented grammar for G as:

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON		
2	c_expression	→	CONSTANT		
3		|	IDENTIFIER		
4		|	c_expression	OPERATOR	CONSTANT
5		|	c_expression	OPERATOR	IDENTIFIER
6	**c_statement′**	→	**c_statement**		

In step 2, we take the augmented grammar as the input and construct a canonical collection of sets of items. We begin by seeding the start state with

I_0= {c_statement′ → • c_statement}

Applying closure on the above set, would get us the items with all the productions that have the start symbol as the left-hand side, namely,

c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON

Applying closure does not yield any extra items since the symbol to the right of the dot is a terminal. Thus, there are two items in the item set for the start state

I_0 = c_statement′ → • c_statement
c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON

This is the first set in canonical collection of sets of LR(0) Items. The canonical collection is now

C = {I_0}

We now perform goto operation on I_0 for all the symbols in the grammar namely SEMI_COLON, EQ_TO_OP, IDENTIFIER, CONSTANT, OPERATOR, c_expression and c_statement. This is shown below.

I_0	=	c_statement′ → • c_statement c_statement → • IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
goto(I_0, IDENTIFIER)	= =	closure set of {c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON} c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON **This is I_1**
goto(I_0, c_statement)	= =	closure set of {c_statement′ → c_expression •} c_statement’ → c_statement • **This is I_2**

goto(I_0, SEMI_COLON)
goto(I_0, EQ_TO_OP)
goto(I_0, OPERATOR)
goto(I_0, CONSTANT)
goto(I_0, c_expression)
		All of these are empty
		Since there is no item in I_0 with SEMI_COLON or EQ_TO_OP or OPERATOR or c_expression before the dot

The canonical collection is now

C = {I_0, I_1, I_2}

We now perform goto operation on the newly added sets of items I_1 and I_2 for all the symbols in the grammar, namely SEMI_COLON, EQ_TO_OP, IDENTIFIER, CONSTANT, OPERATOR, c_expression and c_statement.

I_1	=	c_statement → IDENTIFIER • EQ_TO_OP c_expression SEMI_COLON
goto(I_1, EQ_TO_OP)	= =	closure set of {c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON} c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON **This is I_3**

		c_expression → • CONSTANT
		c_expression → • IDENTIFIER
		c_expression → • c_expression OPERATOR CONSTANT
		c_expression → • c_expression OPERATOR IDENTIFIER
goto(I_1, SEMI_COLON)		
goto(I_1, CONSTANT)		
goto(I_1, IDENTIFIER)		All of these are empty
goto(I_1, OPERATOR)		Since there is no item in I_1 with SEMI_COLON or CONSTANT or
goto(I_1, c_expression)		IDENTIFIER or OPERATOR or c_expression or c_statement before the dot
goto(I_1, c_statement)		

The canonical collection is now

$$C = \{I_0, I_1, I_2, I_3\}$$

I_2	=	c_statement′ → c_statement •
goto(I_2, SEMI_COLON)	=	
goto(I_2, EQ_TO_OP)		All of these are empty
goto(I_2, IDENTIFIER)		Since there is no item in I_2 with SEMI_COLON or EQ_TO_OP or
goto(I_2, CONSTANT)		IDENTIFIER or CONSTANT or OPERATOR or c_expression or
goto(I_2, OPERATOR)		c_statement before the dot
goto(I_2, c_expression)		
goto(I_2, c_statement)		

I_3	=	c_statement → IDENTIFIER EQ_TO_OP • c_expression SEMI_COLON
		c_expression → • CONSTANT
		c_expression → • IDENTIFIER
		c_expression → • c_expression OPERATOR CONSTANT
		c_expression → • c_expression OPERATOR IDENTIFIER
goto(I_3, SEMI_COLON)		
goto(I_3, EQ_TO_OP)		All of these are empty
goto(I_3, OPERATOR)		Since there is no item in I_3 with SEMI_COLON or EQ_TO_OP or
goto(I_3, c_statement)		OPERATOR or c_statement before the dot
goto(I_3, CONSTANT)	=	closure set of {c_expression → CONSTANT •}
	=	c_expression → CONSTANT •
		This is I_4
goto(I_3, IDENTIFIER)	=	closure set of {c_expression → IDENTIFIER •}
	=	{c_expression → IDENTIFIER •} **This is I_5**

goto(I_3, c_expression)	=	closure set of
		{
		c_statement → IDENTIFIER EQ_TO_OP c_expression • SEMI_COLON
		c_expression → c_expression • OPERATOR CONSTANT
		c_expression → c_expression • OPERATOR IDENTIFIER
		}

	=	c_statement IDENTIFIER EQ_TO_OP c_expression • SEMI_COLON **This is I_6** c_expression → c_expression • OPERATOR CONSTANT c_expression → c_expression • OPERATOR IDENTIFIER

The canonical collection is now

$$C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6\}$$

I_4	=	c_expression → CONSTANT •

goto(I_4, SEMI_COLON)
goto(I_4, EQ_TO_OP)
goto(I_4, IDENTIFIER)
goto(I_4, CONSTANT)
goto(I_4, OPERATOR)
goto(I_4, c_expression)
goto(I_4, c_statement)
	=	All of these are empty
		Since there is no item in I_4 with SEMI_COLON or EQ_TO_OP or IDENTIFIER or CONSTANT or OPERATOR or c_expression or c_ statement before the dot

I_5	=	c_expression → IDENTIFIER •

goto(I_5, SEMI_COLON)
goto(I_5, EQ_TO_OP)
goto(I_5, IDENTIFIER)
goto(I_5, CONSTANT)
goto(I_5, OPERATOR)
goto(I_5, c_expression)
goto(I_5, c_statement)
	=	All of these are empty
		Since there is no item in I_5 with SEMI_COLON or EQ_TO_OP or IDENTIFIER or CONSTANT or OPERATOR or c_expression or
		c_statement before the dot

I_6	=	c_statement → IDENTIFIER EQ_TO_OP c_expression • SEMI_COLON c_expression → c_expression • OPERATOR CONSTANT c_expression → c_expression • OPERATOR IDENTIFIER
goto(I_6, SEMI_COLON)	=	closure set of {c_statement → IDENTIFIER = c_expression SEMI_COLON •}

		c_statement : IDENTIFIER = c_expression SEMI_COLON • **This is I_7**

goto(I_6, OPERATOR)		= closure set of
		{
		c_expression → c_expression OPERATOR • CONSTANT
		c_expression → c_expression OPERATOR • IDENTIFIER
		}

	=	c_expression → c_expression OPERATOR • CONSTANT **This is I_8** c_expression → c_expression OPERATOR • IDENTIFIER

The canonical collection is now

$C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8\}$

I_7	=	c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON •

goto(I_7, SEMI_COLON)
goto(I_7, EQ_TO_OP)
goto(I_7, IDENTIFIER)
goto(I_7, CONSTANT)
goto(I_7, OPERATOR)
goto(I_7, c_expression)
goto(I_7, c_statement)
	=	All of these are empty
		Since there is no item in ***I_7*** with SEMI_COLON or EQ_TO_OP or IDENTIFIER or CONSTANT or OPERATOR or c_expression or c_statement before the dot

I_8	=	c_expression → c_expression OPERATOR • CONSTANT c_expression → c_expression OPERATOR • IDENTIFIER
goto(I_8, CONSTANT)	= =	closure set {c_expression → c_expression OPERATOR CONSTANT •} c_expression → c_expression OPERATOR CONSTANT • **This is I_9**
goto(I_8, IDENTIFIER)	= =	closure set {c_expression → c_expression OPERATOR IDENTIFIER •} c_expression → c_expression OPERATOR IDENTIFIER • **This is I_{10}**

The canonical collection is now

$C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}\}$

I_9	=	c_expression → c_expression OPERATOR CONSTANT •

goto(I_9, SEMI_COLON)
goto(I_9, EQ_TO_OP)
goto(I_9, IDENTIFIER)
goto(I_9, CONSTANT)
	=	All of these are empty
		Since there is no item in ***I_9,*** with SEMI_COLON or EQ_TO_OP or IDENTIFIER or CONSTANT or OPERATOR or c_expression or c_ statement before the dot

goto(I_9, OPERATOR)
goto(I_9, c_expression)
goto(I_9, c_statement)		

I_{10}	=	c_expression → c_expression OPERATOR IDENTIFIER •

goto(I_{10}, SEMI_COLON)
goto(I_{10}, EQ_TO_OP)
goto(I_{10}, IDENTIFIER)
goto(I_{10}, CONSTANT)
goto(I_{10}, OPERATOR)
goto(I_{10}, c_expression)
goto(I_{10}, c_statement)
	=	All of these are empty
		Since there is no item in ***I*₁₀** with SEMI_COLON or EQ_TO_OP or IDENTIFIER or CONSTANT or OPERATOR or c_expression or c_statement before the dot

We have performed goto operation on the entire Item sets I_0 through I_{10}, there are no more sets of items generated, and we halt the algorithm.

The final canonical collection is

$$C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}\}$$

Each one of these sets represents the corresponding state. For example, I_0 represents state 0, I_1 represents state 1, etc.

We now start ***step 3***, which is the generation of parsing table from the canonical collection of sets of items.

#	Parsing Table Entries	Explanation

I_0	action_table[0][IDENTIFIER]=s1
	goto_table[0][c_statement]=2
		We saw earlier that goto(I_0, IDENTIFIER) = I_1, hence by Rule 1 of Table 3.26
		We saw earlier that goto(I_0, c_statement) = I_2, hence by Rule 4 of Table 3.26

I_1	action_table[1][EQ_TO_OP]=s3	We saw earlier that goto(I1, IDENTIFIER) = I_3, hence by Rule 1 of Table 3.26
I_2	action_table[2][$]=accept	In Item set I_2, we see the item c_statement → c_statement •, hence by Rule 3 of Table 3.26

I_3	action_table[3][CONSTANT]=s4
	action_table[3][IDENTIFIER]=s5
	goto_table[3][c_expression]=6
		We saw earlier that goto(I_3, CONSTANT) = I_4, hence by Rule 1 of Table 3.26
		We saw earlier that goto(I_3, IDENTIFIER) = I_5, hence by Rule 1 of Table 3.26
		We saw earlier that goto(I_3, IDENTIFIER) = I_6, hence by Rule 4 of Table 3.26

I_4	action_table[4][SEMI_COLON]=r2
	action_table[4][OPERATOR]=r2
		In I_4, there is an item c_expression → CONSTANT • which tells us that action should be to reduce by the production number 2, c_expression → CONSTANT for all elements in FOLLOW(c_ expression), i.e. {SEMI_COLON, OPERATOR}, hence by Rule 2 of Table 3.26

I_5	action_table[5][SEMI_COLON]=r3
	action_table[5][OPERATOR]=r3
		In I_5, there is an item c_expression → IDENTIFIER • which tells us that action should be reduce by the Production 3, c_expression → IDENTIFIER for all elements in FOLLOW(c_expression) i.e. {SEMI_COLON, OPERATOR}, hence by Rule 2 of Table 3.26

I_6	action_table[6][SEMI_COLON]=s7
	action_table[6][OPERATOR]=s8
		We saw earlier that goto(I_6, SEMI_COLON) = I_7, hence By Rule 1 of Table 3.26
		We saw earlier that goto(I_6, OPERATOR) = I_7, hence by Rule 1 of Table 3.26

I_7	action_table[7][$]=r1	In I_7, there is an item c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON •, which tells us that action should be to reduce by the Production number 1, c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON for all elements in FOLLOW(c_statement), i.e. {$} hence by Rule 2 of Table 3.26

I_8	action_table[8][CONSTANT]=s9
	action_table[8][IDENTIFIER]=s10
		We saw earlier that goto(I_8, CONSTANT) = I_9, hence by Rule 1 of Table 3.26
		We saw earlier that goto(I_8, IDENTIFIER) = I_{10}, hence by Rule 1 of Table 3.26

I_9	action_table[9][SEMI_COLON]=r4
	action_table[9][OPERATOR]=r4
		In I_9, there is an item c_expression → c_expression OPERATOR IDENTIFIER •, which tells us that action should be to reduce by the Production number 4, c_expression → c_expression OPERATOR IDENTIFIER for all elements in FOLLOW(c_expression), i.e. {SEMI_COLON, OPERATOR}, hence by Rule 2 of Table 3.26

I_{10}	action_table[10][SEMI_COLON]=r5
	action_table[10][OPERATOR]=r5
		In I_{10}, there is an item c_expression → c_expression OPERATOR CONSTANT •, which tells us that action should be to reduce by the production number 5, c_expression → c_expression OPERATOR CONSTANT for all elements in FOLLOW(c_expression), i.e. {SEMI_COLON, OPERATOR} hence by Rule 2 of Table 3.26

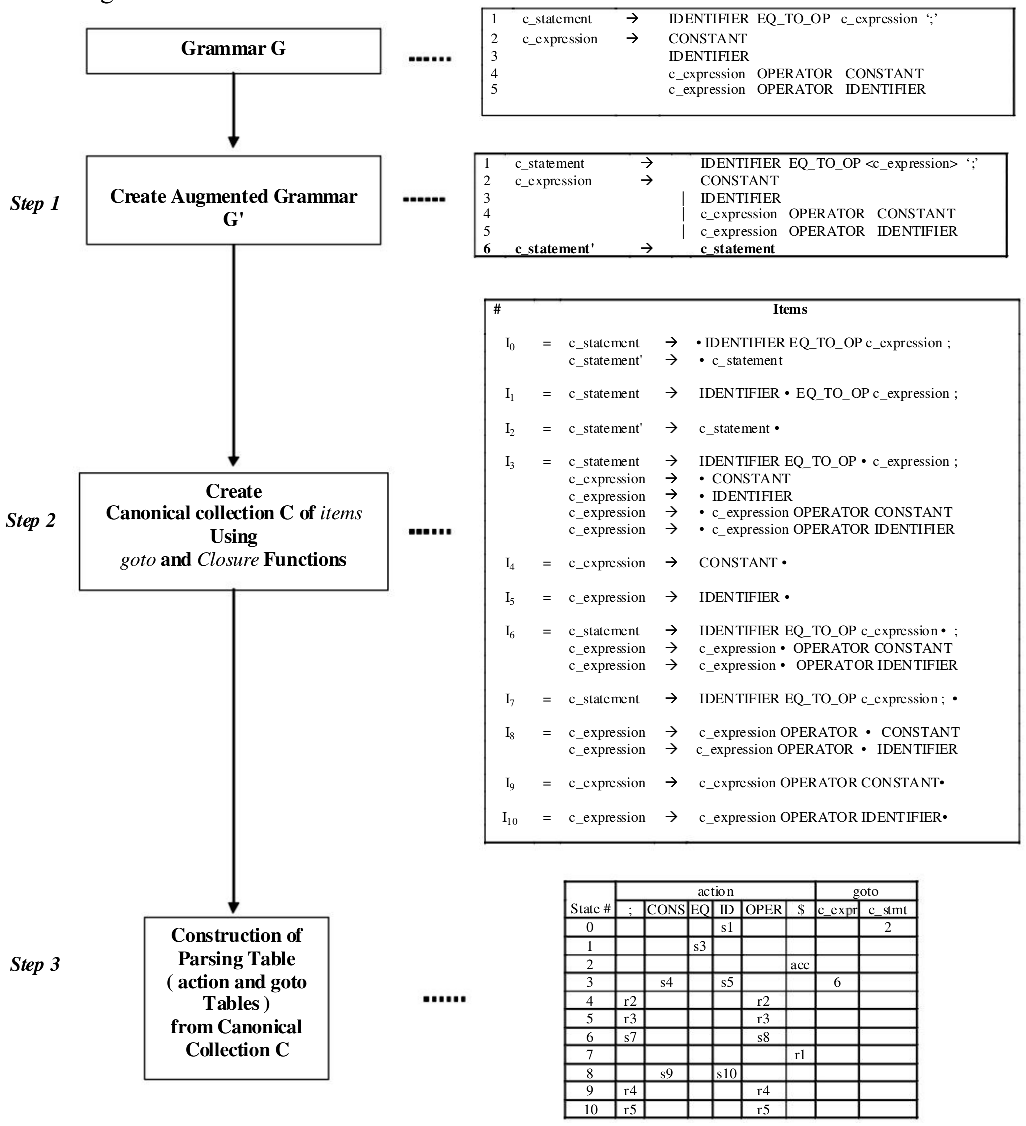
This completes the construction of parsing table entries (see Table 3.27) for the grammar in Table 3.17.

Table 3.27 *Parsing table*

State #	action						goto	
	SEMI_ COLON	CONSTANT	EQ_TO_ OP	IDENTIFIER	OPERATOR	End of Input ($)	c_ expression	c_ statement
0				s1				2
1			s3					
2						accept		
3		s4		s5			6	
4	r2				r2			
5	r3				r3			

6	s7				s8			
7						r1		
8		s9		s10				
9	r4				r4			
10	r5				r5			

Figure 3.14 shows the output at each one of the steps in the construction of parsing table by SLR(1) method for the grammar shown in Table 3.17.


Grammar G
Step 1
Create Augmented Grammar G'
Step 2
Create Canonical collection C of *items* Using *goto* and *Closure* Functions
Step 3
Construction of Parsing Table (action and goto Tables) from Canonical Collection C

1	c_statement	→	IDENTIFIER EQ_TO_OP c_expression ';'
2	c_expression	→	CONSTANT
3			IDENTIFIER
4			c_expression OPERATOR CONSTANT
5			c_expression OPERATOR IDENTIFIER

1	c_statement	→	IDENTIFIER EQ_TO_OP <c_expression> ';'
2	c_expression	→	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR CONSTANT
5		|	c_expression OPERATOR IDENTIFIER
6	**c_statement'**	→	**c_statement**

#		Items		
I_0	=	c_statement	→	• IDENTIFIER EQ_TO_OP c_expression ;
		c_statement'	→	• c_statement
I_1	=	c_statement	→	IDENTIFIER • EQ_TO_OP c_expression ;
I_2	=	c_statement'	→	c_statement •
I_3	=	c_statement	→	IDENTIFIER EQ_TO_OP • c_expression ;
		c_expression	→	• CONSTANT
		c_expression	→	• IDENTIFIER
		c_expression	→	• c_expression OPERATOR CONSTANT
		c_expression	→	• c_expression OPERATOR IDENTIFIER
I_4	=	c_expression	→	CONSTANT •
I_5	=	c_expression	→	IDENTIFIER •
I_6	=	c_statement	→	IDENTIFIER EQ_TO_OP c_expression • ;
		c_expression	→	c_expression • OPERATOR CONSTANT
		c_expression	→	c_expression • OPERATOR IDENTIFIER
I_7	=	c_statement	→	IDENTIFIER EQ_TO_OP c_expression ; •
I_8	=	c_expression	→	c_expression OPERATOR • CONSTANT
		c_expression	→	c_expression OPERATOR • IDENTIFIER
I_9	=	c_expression	→	c_expression OPERATOR CONSTANT•
I_{10}	=	c_expression	→	c_expression OPERATOR IDENTIFIER•

	action						goto	
State #	;	CONS	EQ	ID	OPER	$	c_expr	c_stmt
0				s1				2
1			s3					
2						acc		
3		s4		s5			6	
4	r2				r2			
5	r3				r3			
6	s7				s8			
7						r1		
8		s9		s10				
9	r4				r4			
10	r5				r5			

Fig. 3.14 *Construction of parsing table for the C-statement grammar by SLR(1) method*

Example 9—Construction of Parsing Table by SLR(1) Method This section shows a program constructing the parsing table from grammar by using the SLR(1) method. This takes a grammar as input and outputs the parsing table entries. It uses the 3 steps outlined earlier for constructing the parsing table entries from a given grammar.

```
# Building ex9 Binary
$ g++ -g -Wall grammar.cc slr.cc item.cc ex9.cc -o ex9

# A sample grammar file - sample4.gram
$ cat sample4.gram
c_statement : IDENTIFIER EQ_TO_OP c_expression ;
c_expression : CONSTANT
c_expression : IDENTIFIER
c_expression : c_expression OPERATOR CONSTANT
c_expression : c_expression OPERATOR IDENTIFIER

$ ./ex9 'sample4.gram'
**** FIRST and FOLLOW sets ****
c_expression FIRST={CONSTANT,IDENTIFIER} FOLLOW={;,OPERATOR}
c_statement FIRST={IDENTIFIER} FOLLOW={$}

********* Canonical Collection *********
No of sets in Canonical Collection=11
I(0)  = c_statement : .IDENTIFIER EQ_TO_OP c_expression ;
        c_statementDASH : .c_statement

I(1)  = c_statement : IDENTIFIER .EQ_TO_OP c_expression ;

I(2)  = c_statementDASH : c_statement.

I(3)  = c_statement : IDENTIFIER EQ_TO_OP .c_expression ;
        c_expression : .CONSTANT
        c_expression : .IDENTIFIER
        c_expression : .c_expression OPERATOR CONSTANT
        c_expression : .c_expression OPERATOR IDENTIFIER

I(4)  = c_expression : CONSTANT.

I(5)  = c_expression : IDENTIFIER.

I(6)  = c_statement : IDENTIFIER EQ_TO_OP c_expression .;
        c_expression : c_expression .OPERATOR CONSTANT
        c_expression : c_expression .OPERATOR IDENTIFIER

I(7)  = c_statement : IDENTIFIER EQ_TO_OP c_expression ;.

I(8)  = c_expression : c_expression OPERATOR .CONSTANT
       c_expression : c_expression OPERATOR .IDENTIFIER

I(9)  = c_expression : c_expression OPERATOR CONSTANT.
```

```
I(10) = c_expression : c_expression OPERATOR IDENTIFIER.
-------------------------------------------------------------------------------------
State  | ; CONSTANT  EQ_TO_OP   IDENTIFIER  OPERATOR  $  c_expression   c_statement
-------------------------------------------------------------------------------------
000    |                            s1                                       2
001    |                s3
002    |                                             acc
003    |      s4                    s5                         6
004    | r2                                    r2
005    | r3                                    r3
006    | s7                                    s8
007    |                                              r1
008    |      s9                   s10
009    | r4                                    r4
010    | r5                                    r5
-------------------------------------------------------------------------------------
```

3.5.5.4 ***Construction of Parsing Table by Canonical LR Method*** The SLR method can be used to construct the parsing table for non-ambiguous grammars. However, there are certain grammars for which SLR method does not work despite being non-ambiguous.

Let's take a look at why SLR method fails for certain grammars despite being non-ambiguous. Consider a grammar whose parsing table entries are constructed by using the SLR method.

Table 3.28 *Context-free grammar*

1	s	→	D C a
2	s	→	D a B
3	a	→	C

The sets of LR(0) Items for the above grammar can be found by executing the Example 9 binary.

```
# A sample grammar file - sample7.gram
$ cat sample7.gram
s : D C A
s : D a B
a : C

$ ./ex9 'sample7.gram'
**** FIRST and FOLLOW sets ****
a FIRST={C} FOLLOW={B}
s FIRST={D} FOLLOW={$}

********* Canonical Collection *********
No of sets in Canonical Collection=7
I(0)  =  s : .D C A
         s : .D a B
         sDASH : .s

I(1)  =  s : D .C A
```

```
        s : D .a B
        a : .C

I(2)  = sDASH : s.

I(3)  = s : D C .A
        a : C.

I(4)  = s : D a .B

I(5)  = s : D C A.

I(6)  = s : D a B.
-------------------------------------------
State   |    A    B    C    D    $    a    s
-------------------------------------------
000     |                  s1              2

001     |             s3             4

002     |                       acc

003     |   s5   r3

004     |        s6

005     |                       r1

006     |                       r2
-------------------------------------------
```

The state diagram for the same grammar is shown in Fig. 3.15.

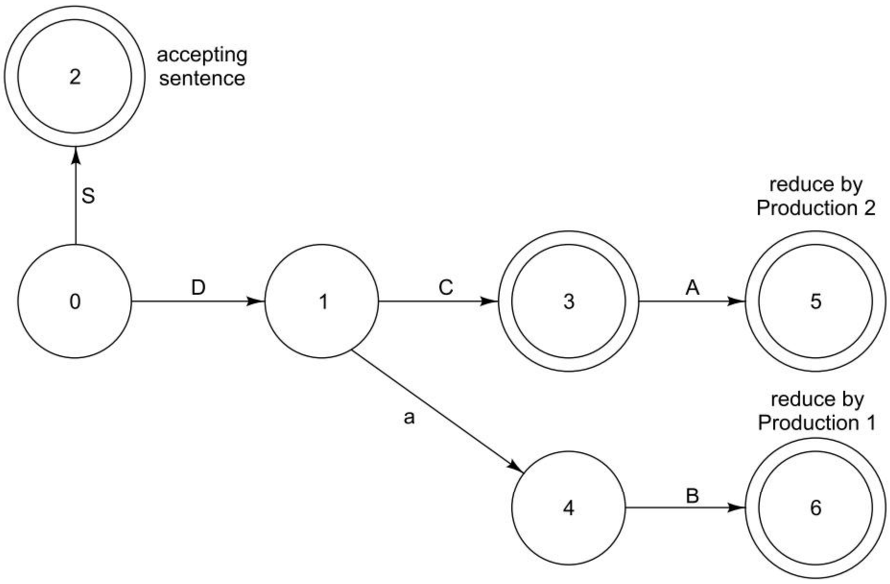


Fig. 3.15 *State diagram for the grammar*

Let's take a look at I_3

I_3 : s → DC • A *Item 1*

a → C • *Item 2*

From the Item 1, we know that if the next input symbol is 'A', a shift is performed to reach state 5. From Item 2, we know that, if the next input symbol is a part of FOLLOW(a), i.e. {'B'}, We reduce by the production a → C. The output also shows action_table[3][A] = s5 and action_table[3][B] = r3.

Let's add an additional production to the grammar as follows:

```
4   s   →   a A
```

Let's execute the Example 9 again on this new grammar.

```
# A sample grammar file - sample7a.gram
$ cat sample7a.gram
s : D C A
s : D a B
a : C
s : a A

$ ./ex9 'sample7a.gram'
$ ./ex9 'sample7a.gram'
**** FIRST and FOLLOW sets ****
a FIRST={C} FOLLOW={A, B}
s FIRST={C, D} FOLLOW={$}

********* Canonical Collection *********
No of sets in Canonical Collection=10
I(0)  =  s : .D C A
         s : .D a B
         a : .C
         s : .a A
         sDASH : .s

I(1)  =  a : C.

I(2)  =  s : D .C A
         s : D .a B
         a : .C

I(3)  =  s : a .A

I(4)  =  sDASH : s.

I(5)  =  s : D C .A
         a : C.

I(6)  =  s : D a .B
```

```
I(7)  =  s : a A.

I(8)  =  s : D C A.

I(9)  =  s : D a B.

Conflicting entry at action_table[5][A] oldval=8 newval=–3
```

This is creating a conflict in the generation of parsing table at the state 5.

Let's take a look at I_5

I_5	:	$s \rightarrow DC \bullet A$	*Item 1*
		$a \rightarrow C \bullet$	*Item 2*

The items in I_5 are exactly same as what we had in I_3 before the addition of the new production. Looking at item 1, in this state tells us that if the next input is A, then a ***shift*** needs to be performed. Item 2 tells us that a ***reduce*** by $a \rightarrow C$ would be performed, if the next input symbol is A, since FOLLOW(a)={A, B}. Thus, if the next input symbol is A, item 1 dictates that there would be a ***shift***, while Item 2 indicates that there should be a ***reduce***. This is the genesis of the conflict.

Actually, the parser should have gone ahead with the shift, when the input is A, because the only possibility of the input being accepted is $s \rightarrow D\ C\ A$. The reduction $a \rightarrow C$ should never be made in this state, when the input is A, since there is no production accepting an input string by 'D a A', even if it were followed by any favourable tokens.

This problem is overcome in the canonical LR and LALR methods of constructing a parsing table. The canonical LR and LALR methods of constructing parsing table entries use a more powerful enhanced item packed with more information that will allow us to rule out some of these invalid reductions like the one '$a \rightarrow C$' above. The enhanced item includes a terminal in addition to the production and dot. Recall that the LR(0) Item can be represented by a 2-tuple of (production, position of dot). The enhanced item called as ***LR(1) item*** consists of a terminal X in addition to the production and position of dot. The LR(1) item is defined by ***(production, position of dot, terminal).*** The terminal X is called the ***look ahead symbol***.

To get an idea on the LR(1) items, Let's take a grammar, one of whose productions is say,

$y \rightarrow a\ B\ c\ D$, where B and D are terminals.

Some of the LR(1) Items possible with this production are

[Y	$\rightarrow$	$a \bullet B\ c\ D$, B]	*Item 1*
[Y	$\rightarrow$	$a \bullet B\ c\ D$, D]	*Item 2*
[Y	$\rightarrow$	$a\ B \bullet c\ D$, B]	*Item 3*
[Y	$\rightarrow$	$a\ B \bullet c\ D$, D]	*Item 4*

The two related procedures, namely the closure and goto, also undergo a change for supporting LR(1) items. Let's see how they change.

Closure Operation for LR(1) Items If I is a set of LR(1) items for a grammar G, then closure of I, represented by closure(I) can be computed using the following 2 rules:

Rule 1	Every element of I is added to closure of I
Rule 2	If an LR(1) item [$X \rightarrow A \bullet BC$, a] exists in I, and there exists a production $B \rightarrow b_1 b_2..b_n$, then add item [$B \rightarrow \bullet\ b_1 b_2..b_n$, z] where z is a terminal in FIRST(Ca), if it is not already in closure(I). Keep applying this rule until there are no more elements added.

To understand the computation of closure operation of LR(1) items, consider the grammar

Table 3.29 *A context-free grammar*

1	s	→	D C A
2		|	D a B
3	a	→	C
4	s	→	a A

Let's take a set I containing two of LR(1) items

I = {[s → D • C A, \$], [s → D • aB, \$]} where \$ is the end of input.

Table 3.30 *Computation of closure set on LR(1) items*

		Items	Comments
closure(I)	=	{[s → D • C A, \$], [s → D • aB, \$]}	By Rule 1, every element of I is also present in closure of I.
	=	{[s → D • C A, \$], [s → D • aB, \$], [a → • C, B]}	Rule 2 applied on the first item [s → D • CA, \$] does not yield any item, since the symbol after the dot is a terminal (i.e. no productions would exist with the LHS as 'C'). Applying Rule 2 applied on the second item [s → D • aB, \$], we find that there exists a Production 3 (a → C) whose left-hand side is 'a' (the non-terminal after the dot). The FIRST(B\$) is {B} (recall that the FIRST(non-terminal) is itself.). By the rule 2, we have the closure of I containing the element [a : • C, B].
	=	{[s → D • C A, \$], [s → D • aB, \$], [a → • C, B]}	Applying Rule 2 on the entire set of items yield no additional items.

The closure computation for LR(1) Item set is summarised in Algorithm 3.9.

```
I is a set of LR(1) items belonging to grammar G
J is the closure(I)
```

```
J=I
do{
        added=0
        for (each item [X → A · BC, Y] in J) {
             for each production B → $b_1b_2b_3..b_n$ exists in grammar G' {
                  for each terminal Z in FIRST(CY) {
                       if (item [B → · $b_1b_2b_3..b_n$, Z] does not exist in C){
                            add the item B →.$b_1b_2b_3..b_n$, Z
                            added ++
                       }
                  }
             }
        }
} while (added > 0)
```

Algorithm 3.9 *Closure computation on LR(1) item set*

goto Operation for LR(1) Items If I is a set of LR(1) Items which contains an item [A → α • $X\beta$, a] where X is any grammar symbol, then ***goto(I, X)*** is defined as the closure of the set of all the items [A → αX • β, a]

To understand the goto computation on LR(1) items, consider the Grammar in Table 3.29.

Let's take a set of LR(1) Items I with four elements

I = {[s → • DCA, \$] [s → • DaB, \$] [a → • C, A] [s → • aA, \$]}

From the above definition of goto operation on LR(1) items, we have

goto(I, D) = closure {[s → D • CA, \$] [s → D • aB, \$]}

= {[s → D • CA, \$] [s → D • aB, \$] [a → • C, B]} from example in Table 3.30.

= {[s → D • CA, \$], [s → D • aB, \$], [a → • C, B]}

The goto(I, X) computation is formalised in Algorithm 3.10.

I, J, tmp_set are sets of items
J = goto(I, X)

```
for (each item [A → α · Xβ, a] in I)

        Add the item [A → α · β, a] to tmp_set
}
J= closure(tmp_set)
```

Algorithm 3.10 *goto(I, X) for a set I of LR(1) Items*

Armed with the concept of LR(1) Item, methods to perform closure operation and goto operations on LR(1) items, we shall try to understand the canonical LR parsing method for constructing a parsing table.

Procedure for constructing the Parsing Table Let's revisit the steps shown in Fig. 3.11 with reference to canonical LR method of constructing the parsing table.

In step 1, we construct the augmented grammar (G′) by adding an additional production S' → S where S is the start symbol of the original grammar (G).

Step 2 involves creation of a canonical collection of sets of LR(1) items for LR(1) parsing table. We start with the Item set 0 (I_0) containing the LR(1) item [S′ → S,\$] and the closure obtained on it. This is the first set added in the canonical collection of sets C. Now, for every set I_n present in canonical collection C, we determine goto(I_n, X), for every symbol X in the grammar, if that yields a set not already present in the canonical collection of sets, we add it to the collection. If no sets are added to the collection during one traversal of the entire canonical collection, we halt the algorithm completing the canonical collection of sets of LR(1) items. This procedure is shown in Algorithm 3.7. This algorithm uses the goto and closure functions for the LR(1) Items that were described in the previous section.

I, J, tmp_set are sets of items

```
I = {S'→ · S, $}
J = closure (I)
Add J as one set in canonical collection C /* This corresponds to state 0 */

do
```

```
{
        added = false;
        for ( each set I in canonical collection C)
             for ( each grammar symbol X in symbol table)
                  tmp_set = goto(I,X)
                  if( tmp_set is not present in canonical collection C){
                       added = true;
                       add tmp_set in canonical collection C
                  }
             }
        }
} while ( added == true)
```

Algorithm 3.11 *Creating canonical collection of sets of LR(1) items*

In step 3, we construct the action and goto table using the canonical collection of sets of LR(1) items by using Algorithm 3.12. The following rules form the basis of construction of LR(1) parsing table from the canonical collection of sets of items.

Table 3.31 *Rules for constructing parsing table from canonical collection*

Rule 1	If there is an item [$A \to \alpha \bullet X\beta$, b] in I_i and goto(I_i, X) is in the Item set I_j then action[I][X] = shift j, where X is a terminal
Rule 2	If there is an Item [$A \to \alpha \bullet$, X] in I_i (and $A \neq S'$) set action[i][X] = reduce by $A \to \alpha$
Rule 3	If there is an item [$S' \to S \bullet$, \$] in I_i then set action[i][\$] = accept
Rule 4	If the goto(I_i, X) = I_j then goto[i][X] = j, where X is a non-terminal

All the entries not defined by above rules are error entries. If there is conflict in the entries generated out of the above rules, then the grammar is not LR(1) grammar. The algorithm fails to produce a parser. These rules are formalised in the Algorithm 3.12.

C is canonical collection of sets of Items = {I_0, I_1, I_2,... I_n} created using Algorithm 3.11

```
for (each item set I_i in Canonical Collection) {
        for (each item P in an Item set I_i) {
              if (P is [S'→ S·, $]){
                    action[i][$]=ACCEPT;
              } else if (P is [A → α·, X]){
                    action[i][X] = Reduce by the Production A → α ·
              } else if (P is a item [A → α · Xβ, b] and X is a Terminal){
                     if (goto(I_i,X) == I_j){
                           action[I][X]=shift j ;
                     }
              } else if (P is a item [A → α · Xβ, b] and X is a non-terminal) {
                       if (goto(I_i,X) == I_j){
                             goto[I][X]=j ;
                       }
              }
        }
}
```

Algorithm 3.12 *Construction of parsing table*

Example 10—Construction of Parsing Table by canonical LR Method This section demonstrates an example program that can construct parsing table entries using the canonical parsing table construction method. The program implements Algorithm 3.9, Algorithm 3.10 and Algorithm 3.11 for constructing the parsing table entries.

```
# Building ex10 Binary
$  g++  -g  -Wall  grammar.cc  canonical_lr.cc  lr1item.cc  item.cc  ex10.cc  -o  ex10

# A sample grammar file - sample7a.gram
$ cat sample7a.gram
s : D C A
s : D a B
a : C
s : a A

$ ./ex10 'sample7a.gram'

********* Canonical Collection *********
No of sets in Canonical Collection=10
I(0)  =  [s : .D C A,$]
         [s : .D a B,$]
         [a : .C,A]
         [s : .a A,$]
         [sDASH : .s,$]

I(1)  =  [a : C.,A]

I(2)  =  [s : D .C A,$]
         [s : D .a B,$]
         [a : .C,B]
I(3)  =  [s : a .A,$]
I(4)  =  [sDASH : s.,$]
I(5)  =  [s : D C .A,$]
         [a : C.,B]
I(6)  =  [s : D a .B,$]
I(7)  =  [s : a A.,$]
I(8)  =  [s : D C A.,$]
I(9)  =  [s : D a B.,$]
-----------------------------------------------------
State  |  A     B      C      D      $      a      s
-----------------------------------------------------

000    |               s1     s2            3      4
001    |  r3
002    |               s5                   6
003    |  s7
004    |                            acc
005    |  s8    r3
006    |        s9
007    |                            r4
008    |                            r1
009    |                            r2
-----------------------------------------------------
```

3.5.5.5 *Construction of Parsing Table by Look Ahead LR Method* The look ahead LR method for constructing a parsing table is an improvisation on the canonical LR parsing table construction method. The LALR method for constructing parsing table uses the LR(1) item, like the canonical LR method. Most of the popular parser generator programs like 'bison', 'yacc', etc. use the LALR method for constructing the parsing table.

Consider the grammar in Table 3.32, which describes a simple selection statement of the form say (*count* = = *20*), where count is a variable.

Table 3.32 *Grammar for selection statement*

1	sel_stmt	→	cond_expr EQTO cond_expr
2	cond_expr	→	(cond_expr)
3		|	pri_expr
4	pri_expr	→	IDENTIFIER
		|	CONSTANT

We have avoided the full description of sel_stmt, which could contain less than/greater than operators, cast expression, etc. The idea is to explore a few key concepts by keeping it simple.

Let's see how the canonical LR parsing works on this grammar by executing Example 10.

```
# A sample grammar file - sample9.gram
$ cat sample9.gram
sel_stmt : cond_expr EQTO cond_expr
cond_expr : ( cond_expr)
cond_expr : pri_expr
pri_expr : ID
pri_expr : CONSTANT

$ ./ex10 'sample9.gram'

********* Canonical Collection *********
No of sets in Canonical Collection=23
I(0)   = [sel_stmt : .cond_expr EQTO cond_expr,$]
        [cond_expr : .( cond_expr),EQTO]
        [cond_expr : .pri_expr,EQTO]
        [pri_expr : .ID,EQTO]
        [pri_expr : .CONSTANT,EQTO]
        [sel_stmtDASH : .sel_stmt,$]

I(1)   = [cond_expr : .( cond_expr),)]
        [cond_expr : ( .cond_expr),EQTO]
        [cond_expr : .pri_expr,)]
        [pri_expr : .ID,)]
        [pri_expr : .CONSTANT,)]

I(2)   = [pri_expr : CONSTANT.,EQTO]
```

```
I(3)  = [pri_expr : ID.,EQTO]

I(4)  = [sel_stmt : cond_expr .EQTO cond_expr,$]

I(5)  = [cond_expr : pri_expr.,EQTO]

I(6)  = [sel_stmtDASH : sel_stmt.,$]

I(7)  = [cond_expr : .( cond_expr),)]
        [cond_expr : ( .cond_expr),)]
        [cond_expr : .pri_expr,)]
        [pri_expr : .ID,)]
        [pri_expr : .CONSTANT,)]

I(8)  = [pri_expr : CONSTANT.,)]

I(9)  = [pri_expr : ID.,)]

I(10) = [cond_expr : ( cond_expr .),EQTO]

I(11) = [cond_expr : pri_expr.,)]

I(12) = [sel_stmt : cond_expr EQTO .cond_expr,$]
        [cond_expr : .( cond_expr),$]
        [cond_expr : .pri_expr,$]
        [pri_expr : .ID,$]
        [pri_expr : .CONSTANT,$]

I(13) = [cond_expr : ( cond_expr .),)]

I(14) = [cond_expr : ( cond_expr).,EQTO]

I(15) = [cond_expr : .( cond_expr),)]
        [cond_expr : ( .cond_expr),$]
        [cond_expr : .pri_expr,)]
        [pri_expr : .ID,)]
        [pri_expr : .CONSTANT,)]

I(16) = [pri_expr : CONSTANT.,$]

I(17) = [pri_expr : ID.,$]

I(18) = [sel_stmt : cond_expr EQTO cond_expr.,$]

I(19) = [cond_expr : pri_expr.,$]

I(20) = [cond_expr : ( cond_expr).,)]

I(21) = [cond_expr : ( cond_expr .),$]

I(22) = [cond_expr : ( cond_expr).,$]
```

State	(	)	CONSTANT	EQTO	ID	$	cond_expr	pri_expr	sel_stmt
000	s1		s2		s3		4	5	6
001	s7		s8		s9		10	11	
002				r5					
003				r4					
004				s12					
005				r3					
006						acc			
007	s7		s8		s9		13	11	
008		r5							
009		r4							
010		s14							
011		r3							
012	s15		s16		s17		18	19	
013		s20							
014				r2					
015	s7		s8		s9		21	11	
016						r5			
017						r4			
018						r1			
019						r3			
020		r2							
021		s22							
022						r2			

The canonical LR method of constructing the parsing table has generated 23 states. For a grammar with few rules like the above one, the number of states is pretty reasonable. But, for a full-fledged grammar (e.g. C language grammar) the number of states produced by canonical LR(1) method is typically in thousands. Thus to store the parsing table, the parser would necessitate enormous amount of memory. The LALR method of constructing the parsing table alleviates the problem of parser needing a large amount of memory.

Let's follow the grammar of Table 3.32 and study the LR(1) Items for each of the states as generated by the canonical LR method given in Example 10.

```
I(0) :[ sel_stmt : .cond_expr EQTO cond_expr,$ ]
      [ cond_expr : .( cond_expr ),EQTO ]
      [ cond_expr : .pri_expr,EQTO ]
      [ pri_expr : .ID,EQTO ]
      [ pri_expr : .CONSTANT,EQTO ]
      [ sel_stmtDASH : .sel_stmt,$ ]

I(1)  [ cond_expr : .( cond_expr ),) ]
      [ cond_expr : ( .cond_expr ),EQTO ]
      [ cond_expr : .pri_expr,) ]
      [ pri_expr : .ID,) ]
      [ pri_expr : .CONSTANT,) ]

I(2)  [ pri_expr : CONSTANT. ,EQTO ]

I(3)  [ pri_expr : ID. ,EQTO ]
```

```
I(8)  [ pri_expr : CONSTANT. ,) ]

I(9)  [ pri_expr : ID. ,) ]

I(10) [ cond_expr : ( cond_expr .),EQTO ]

I(11) [ cond_expr : pri_expr. ,) ]

I(12) [sel_stmt:cond_expr EQTO.cond_expr,$]
      [ cond_expr : .( cond_expr ),$ ]
      [ cond_expr : .pri_expr,$ ]
      [ pri_expr : .ID,$ ]
      [ pri_expr : .CONSTANT,$ ]

I(13) [ cond_expr : ( cond_expr .),) ]
```

```
I(4)   [ sel_stmt : cond_expr .EQTO cond_expr,$ ]

I(5)   [ cond_expr : pri_expr. ,EQTO ]

I(6)   [ sel_stmtDASH : sel_stmt. ,$ ]

I(7)   [ cond_expr : .( cond_expr ),) ]
       [ cond_expr : ( .cond_expr ),) ]
       [ cond_expr : .pri_expr,) ]
       [ pri_expr : .ID,) ]
       [ pri_expr : .CONSTANT,) ]
```

```
I(14) [ cond_expr : ( cond_expr ). ,EQTO ]

I(15) [ cond_expr : .( cond_expr ),) ]
      [ cond_expr : ( .cond_expr ),$ ]
      [ cond_expr : .pri_expr,) ]
      [ pri_expr : .ID,) ]
      [ pri_expr : .CONSTANT,) ]

I(16) [ pri_expr : CONSTANT. ,$ ]

I(17) [ pri_expr : ID. ,$ ]

I(18) [sel_stmt:cond_expr EQTOcond_expr.,$]

I(19) [ cond_expr : pri_expr. ,$ ]

I(20) [ cond_expr : ( cond_expr ). ,) ]

I(21) [ cond_expr : ( cond_expr .),$ ]

I(22) [ cond_expr : ( cond_expr ). ,$ ]
```

Let's take a look at the LR(1) Item sets I_{14}, I_{20}, I_{22}.These three item sets have a similarity, all of them have

cond_expr : (cond_expr) •

as the first part of the LR(1) element, only the look ahead symbol varies. In I_{14} the look ahead is EQTO symbol, in I_{20} the look ahead is ')' symbol, while in I_{22} the look ahead is '$', the end-of-input symbol. The three LR(1) item sets represent states 14, 20 and 22 where the '(', cond_expr and ')' have already been consumed. The corresponding rows in action and goto table for the three states 14, 20 and 22 are highlighted in Table 3.33.

Table 3.33 *Canonical parsing table for the grammar in Table 3.32*

TOP OF STACK (Current State)	action							goto	
	IDENTIFIER	CONSTANT	(EQ_TO_OP)			End of Input ($)	cond_expr	pri_expr	sel_stmt
0	s3	s2	s1				4	6	5
1	s9	s8	s7				10	11	
2				r5					
3				r4					
4				s12					
5				r3					
6						accept			
7	s9	s8	s7				13	11	
8					r5				
9					r4				
10					s14				

(*Contd*)

Table 3.33 (*contd*)

11					r3				
12	s17	s16	s15				18	19	
13					s20				
14				r2					
15	s9	s8	s7				21	11	
16						r5			
17						r4			
18						r1			
19						r3			
20					r2				
21					s22				
22						r2			

The sets containing the same first components of the LR(1) Items are said to have the same ***core***. We just saw that the states 14, 20 and 22 have the same core which is *cond_expr : (cond_expr)* •

In LALR method of constructing the parsing table, the LR(1) Item sets with same core are merged to form a single Item set. Since each one of the LR(1) Item sets represent a state, we can see in the above example that the three states 14, 20 and 22 would be merged into one state. This merged state (let's call it as $s_{14\text{-}20\text{-}22}$) would be entered in situations where any one of the constituents would have been originally entered. For example, an input of ')' in state 10 would have forced a transition to state 14 originally. Now the same would force a transition to the state $s_{14\text{-}20\text{-}22}$. The transition from this merged state $s_{14\text{-}20\text{-}22}$ would be determined by the constituent states reaction to the next input symbol. For example, the merged state $s_{14\text{-}20\text{-}22}$ would be reducing by Production 2 on receipt EQ_TO_OP as the next input symbol. This was determined by the constituent state 14. On similar lines the merged state $s_{14\text{-}20\text{-}22}$ would be reducing by Production 2 on receipt ')' as the next input symbol as determined by the constituent state 20. It can be proved that the merged state can never have a shift-shift or a shift-reduce conflict due to the constituent states with respect to any of the input symbol, i.e. it can never happen that on the receipt of a given input symbol X, one of the constituent states would have a 'shift j' and another constituent state would have a 'shift k' or 'reduce n' for the same input symbol.

Coming back to the discussion on the LR(1) item sets generated for the grammar in Table 3.32, we can also see that the states 2, 8 and 16 having a common first part of LR(1) in pri_expr : CONSTANT• . They can be combined to create a new state (called as $s_{2\text{-}8\text{-}16}$), and so on.

This merging of LR(1) Item sets with same core to form condensed LR(1) item sets is the main principle behind the LALR method of constructing the parsing table.

Let's revisit the steps shown in Fig. 3.11 with reference to look ahead LR method of constructing the parsing table.

Step 1: The Augmented grammar is created in the same way as SLR or canonical LR method.

Step 2: The collection C of sets of LR(1) Items is performed in the same way as canonical LR method using Algorithm 3.11. Let's say C contains the sets of LR(1) Items $\{I_0, I_1, I_2, I_3 \ldots I_n\}$. After the construction of C, we identify all the sets having the same core and merge them. This results in a collection of condensed sets of LR(1) Items—C′ containing $\{J_0, J_1, J_2, J_3 \ldots J_k\}$, where J_i is a union of one or more sets of LR(1) Item sets having the same core, denoted mathematically as $J_i = \{I_1 \text{ U } I_2 \text{ U } I_3 \ldots \text{ U } I_k\}$

Step 3: We construct the action and goto table using the original and condensed collection of sets of LR(1) items by using Algorithm 3.13, which is an improvisation on the Algorithm 3.12. The fundamental change in the Algorithm 3.13 compared to Algorithm 3.12 lies in the fact that the entries into action and goto table are made keeping the index of the condensed item set and not the original Item set.

C is canonical collection of sets of Items = $\{I_0, I_1, I_2,\ldots I_n\}$ created using Algorithm 3.7

```
for ( each item set $I_i$ in canonical collection) {
        let $J_k$ be the condensed set whose constituent is the item set $I_i$
        for ( each item P in a Item set $I_i$) {
             if ( P is [S'→ S ·, $]){
                  action[k][$]=ACCEPT;
             } else if ( P is [A → $\alpha$ · x]){
                  action[k][X] = Reduce by the Production A → $\alpha$ ·
             } else if ( P is a item [A → $\alpha \cdot X\beta$, b] and X is a Terminal){
                  if ( goto($I_i$,X) == $I_j$){
                       let $J_n$ be the condensed set whose constituent is the item set $I_j$
                       action[k][X]=shift n ;
                  }
             } else if (P is an item [A → $\alpha \cdot X\beta$, b] and X is a non-terminal) {
                  if ( goto($I_i$,X) == $I_j$){
                       let $J_n$ be the condensed set whose constituent is the item set $I_j$
                       goto[k][X]=n ;
                  }
             }
        }
}
```

Algorithm 3.13 *Construction of parsing table*

Example 11—Construction of Parsing Table by LALR Method This section demonstrates an example program that can construct parsing table entries using the LALR method. The program implements the Algorithm 3.13 for constructing the parsing table entries.

```
# Building ex11 Binary
$ g++ -g -Wall grammar.cc canonical_lr.cc lalr.cc lr1item.cc item.cc ex11.cc -o ex11

# A sample grammar file - sample9.gram
$ cat sample9.gram
sel_stmt : cond_expr EQTO cond_expr
cond_expr : ( cond_expr)
cond_expr : pri_expr
pri_expr : ID
pri_expr : CONSTANT

$ ./ex11 'sample9.gram'

********* canonical collection *********
No of sets in Canonical Collection=23
I(0)  =  [sel_stmt : .cond_expr EQTO cond_expr,$]
```

```
          [cond_expr : .( cond_expr),EQTO]
          [cond_expr : .pri_expr,EQTO]
          [pri_expr : .ID,EQTO]
          [pri_expr : .CONSTANT,EQTO]
          [sel_stmtDASH : .sel_stmt,$]

I(1)   =  [cond_expr : .( cond_expr),)]
          [cond_expr : ( .cond_expr),EQTO]
          [cond_expr : .pri_expr,)]
          [pri_expr : .ID,)]
          [pri_expr : .CONSTANT,)]
I(2)   =  [pri_expr : CONSTANT.,EQTO]

I(3)   =  [pri_expr : ID.,EQTO]

I(4)   =  [sel_stmt : cond_expr .EQTO cond_expr,$]

I(5)   =  [cond_expr : pri_expr.,EQTO]

I(6)   =  [sel_stmtDASH : sel_stmt.,$]

I(7)   =  [cond_expr : .( cond_expr),)]
          [cond_expr : ( .cond_expr),)]
          [cond_expr : .pri_expr,)]
          [pri_expr : .ID,)]
          [pri_expr : .CONSTANT,)]

I(8)   =  [pri_expr : CONSTANT.,)]

I(9)   =  [pri_expr : ID.,)]

I(10) =   [cond_expr : ( cond_expr .),EQTO]

I(11) =   [cond_expr : pri_expr.,)]

I(12) =   [sel_stmt : cond_expr EQTO .cond_expr,$]
          [cond_expr : .( cond_expr),$]
          [cond_expr : .pri_expr,$]
          [pri_expr : .ID,$]
          [pri_expr : .CONSTANT,$]

I(13) =   [cond_expr : ( cond_expr .),)]

I(14) =   [cond_expr : ( cond_expr).,EQTO]

I(15) =   [cond_expr : .( cond_expr),)]
          [cond_expr : ( .cond_expr),$]
          [cond_expr : .pri_expr,)]
          [pri_expr : .ID,)]
          [pri_expr : .CONSTANT,)]
```

```
I(16) = [pri_expr : CONSTANT.,$]

I(17) = [pri_expr : ID.,$]

I(18) = [sel_stmt : cond_expr EQTO cond_expr.,$]

I(19) = [cond_expr : pri_expr.,$]

I(20) = [cond_expr : ( cond_expr).,)]

I(21) = [cond_expr : ( cond_expr .),$]

I(22) = [cond_expr : ( cond_expr).,$]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
State  |   (    )   CONSTANT  EQTO     ID    $  cond_expr   pri_expr    sel_stmt
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
000    |  s1           s2              s3           4          5          6
001    |  s7           s8              s9           10         11
002    |                        r5
003    |                        r4
004    |                       s12
005    |                        r3
006    |                                   acc
007    |  s7           s8              s9           13         11
008    |       r5
009    |       r4
010    |       s14
011    |       r3
012    |  s15         s16             s17          18         19
013    |       s20
014    |                        r2
015    |  s7           s8              s9           21         11
016    |                                    r5
017    |                                    r4
018    |                                    r1
019    |                                    r3
020    |       r2
021    |       s22
022    |                                    r2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
No of condensed sets = 11
old state=0 new state = 0
old state=1 new state = 1
old state=2 new state = 2
old state=3 new state = 3
old state=4 new state = 4
old state=5 new state = 5
old state=6 new state = 6
old state=7 new state = 1
old state=8 new state = 2
old state=9 new state = 3
old state=10 new state = 7
old state=11 new state = 5
old state=12 new state = 8
old state=13 new state = 7
old state=14 new state = 9
old state=15 new state = 1
old state=16 new state = 2
old state=17 new state = 3
old state=18 new state = 10
old state=19 new state = 5
old state=20 new state = 9
old state=21 new state = 7
old state=22 new state = 9

******Refined Canonical Collection *******
No of sets in condensed collection = 11
I(0)  = [sel_stmt : .cond_expr EQTO cond_expr,$]
        [cond_expr : .( cond_expr ),EQTO]
        [cond_expr : .pri_expr,EQTO]
        [pri_expr : .ID,EQTO]
        [pri_expr : .CONSTANT,EQTO]
        [sel_stmtDASH : .sel_stmt,$]

I(1)  = [cond_expr : .( cond_expr),)]
        [cond_expr : ( .cond_expr),$]
        [cond_expr : ( .cond_expr),)]
        [cond_expr : ( .cond_expr),EQTO]
        [cond_expr : .pri_expr,)]
        [pri_expr : .ID,)]
        [pri_expr : .CONSTANT,)]

I(2)  = [pri_expr : CONSTANT.,$]
        [pri_expr : CONSTANT.,)]
        [pri_expr : CONSTANT.,EQTO]

I(3)  = [pri_expr : ID.,$]
        [pri_expr : ID.,)]
        [pri_expr : ID.,EQTO]

I(4)  = [sel_stmt : cond_expr .EQTO cond_expr,$]

I(5)  = [cond_expr : pri_expr.,$]
        [cond_expr : pri_expr.,)]
        [cond_expr : pri_expr.,EQTO]

I(6)  = [sel_stmtDASH : sel_stmt.,$]

I(7)  = [cond_expr : ( cond_expr .),$]
        [cond_expr : ( cond_expr .),)]
        [cond_expr : ( cond_expr .),EQTO]
```

```
I(8)   = [sel_stmt : cond_expr EQTO .cond_expr,$]
         [cond_expr : .( cond_expr),$]
         [cond_expr : .pri_expr,$]
         [pri_expr : .ID,$]
         [pri_expr : .CONSTANT,$]

I(9)   = [cond_expr : ( cond_expr).,$]
         [cond_expr : ( cond_expr).,)]
         [cond_expr : ( cond_expr).,EQTO]

I(10)  = [sel_stmt : cond_expr EQTO cond_expr.,$]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
State   |   (   )   CONSTANT  EQTO    ID    $    cond_expr    pri_expr    sel_stmt
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
000     |  s1          s2             s3             4            5           6
001     |  s1          s2             s3             7            5
002     |      r5              r5          r5
003     |      r4              r4          r4
004     |                      s8
005     |      r3              r3          r3
006     |                                 acc
007     |      s9
008     |  s1          s2             s3            10            5
009     |      r2              r2          r2
010     |                                  r1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Limitation of Look Ahead LR(1) Method The LALR method for constructing the parsing table works very well for most of the unambiguous grammars. However, it can have a reduce-reduce conflict for some of the grammars that LR(1) method can resolve.

Consider the grammar in Table 3.34.

Table 3.34 *Context-free grammar*

1	s	$\rightarrow$	A a D
2	s	$\rightarrow$	B b D
3	s	$\rightarrow$	A b E
4	s	$\rightarrow$	B a E
5	a	$\rightarrow$	C
6	b	$\rightarrow$	C

Let's execute the program for constructing the parsing table by LALR method shown in Example 11 giving the above grammar as input.

```
# A sample grammar file - sample10.gram
$ cat sample10.gram
S : a A d
S : b B d
S : a B e
S : b A e
A : c
B : c

$ ./ex11 'sample10.gram'

********* canonical collection *********
No of sets in canonical collection=14
I(0)   = [S : .a A d,$]
         [S : .b B d,$]
         [S : .a B e,$]
         [S : .b A e,$]
         [SDASH : .S,$]

I(1)   = [SDASH : S.,$]

I(2)   = [S : a .A d,$]
         [S : a .B e,$]
         [A : .c,d]
         [B : .c,e]

I(3)   = [S : b .B d,$]
         [S : b .A e,$]
         [A : .c,e]
         [B : .c,d]

I(4)   = [S : a A .d,$]

I(5)   = [S : a B .e,$]

I(6)   = [A : c.,d]
         [B : c.,e]

I(7)   = [S : b A .e,$]
I(8)   = [S : b B .d,$]

I(9)   = [A : c. ,e]
         [B : c. ,d]

I(10)  = [S : a A d. ,$]

I(11)  = [S : a B e. ,$]

I(12)  = [S : b A e. ,$]
```

```
I(13)  = [S : b B d.,$]
---------------------------------------------------------------
State   |    a     b     c     d     e     $     A     B     S
---------------------------------------------------------------
000     |    s2    s3                                        1
001     |                                  acc
002     |                s6                      4     5
003     |                s9                      7     8
004     |                     s10
005     |                           s11
006     |                      r5    r6
007     |                           s12
008     |                     s13
009     |                      r6    r5
010     |                                  r1
011     |                                  r3
012     |                                  r4
013     |                                  r2
---------------------------------------------------------------
No of condensed sets = 13
old state=0 new state = 0
old state=1 new state = 1
old state=2 new state = 2
old state=3 new state = 3
old state=4 new state = 4
old state=5 new state = 5
old state=6 new state = 6
old state=7 new state = 7
old state=8 new state = 8
old state=9 new state = 6
old state=10 new state = 9
old state=11 new state = 10
old state=12 new state = 11
old state=13 new state = 12

******Refined canonical collection *******
No of sets in condensed collection = 13
I(0)   = [S : .a A d,$]
         [S : .b B d,$]
         [S : .a B e,$]
         [S : .b A e,$]
         [SDASH : .S,$]

I(1)   = [SDASH : S.,$]

I(2)   = [S : a .A d,$]
         [S : a .B e,$]
         [A : .c,d]
         [B : .c,e]
```

```
I(3)   = [S : b .B d,$]
         [S : b .A e,$]
         [A : .c,e]
         [B : .c,d]

I(4)   = [S : a A .d,$]

I(5)   = [S : a B .e,$]

I(6)   = [A : c.,d]
         [A : c.,e]
         [B : c.,d]
         [B : c.,e]

I(7)   = [S : b A .e,$]

I(8)   = [S : b B .d,$]

I(9)   = [S : a A d.,$]

I(10)  = [S : a B e.,$]

I(11)  = [S : b A e.,$]

I(12)  = [S : b B d.,$]

Conflicting entry at action_table[6][e] oldval=-6 newval=-5
Conflicting entry at action_table[6][d] oldval=-5 newval=-6
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 
State    |     a     b     c     d     e     $     A     B     S
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 
000      |    s2    s3                                         1
001      |                                  acc
002      |                s6                       4     5
003      |                s6                       7     8
004      |                      s9
005      |                           s10
006      |                      r5    r6
007      |                           s11
008      |                     s12
009      |                                  r1
010      |                                  r3
011      |                                  r4
012      |                                  r2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 
```

The LALR method was unable to construct a parsing table for the above grammar due to conflicts. The canonical LR method can generate the parsing table for the same grammar. It proves that, even though the LALR method does improve the efficiency in terms memory consumed for the parsing table, there are cases where the LR(1) method is better.

3.5.5.6 *Comparison of the Parsing Table Construction Methods* We have studied three types of LR parsing table construction methods namely, SLR, LR(1) and LALR methods. Let's make a comparison of these methods.

The following table compares the 3 methods of constructing the parsing table from the grammar with respect to vital parameters.

Method	Usage of look-ahead information	Item entity	goto and closure functions	Grammar it applies to
SLR(1)	Yes – 1 symbol	LR(0) Item	Different from LR(1)	SLR(1) – Sub-set of LR(1) grammar
LR(1)	Yes – 1 symbol	LR(1) Item		LR(1) – Largest class of LR grammar
LALR(1)	Yes – 1 symbol	LR(1) Item	Same as LR(1)	LALR(1) – Sub-set of LR(1) grammar but greater than SLR(1) grammar

The following set diagram depicts the applicability of each of these methods to construct parsing table on grammars. The SLR(1) grammar indicates those grammars for which the SLR(1) method can be used successfully to construct a parsing table, the LALR(1) indicates the grammars for which the LALR(1) method can be successfully used for constructing a parsing table, and so on. From Fig. 3.16, it is clear that the LR(1) method of constructing the parsing table can be used on a majority of non-ambiguous grammars. The LALR(1) method can be applied on a sub-set of LR(1) grammars. The SLR(1) method can be used for constructing a parsing table on a sub-set of LALR(1) grammars. Clearly the LR(1) method can be used to construct parsing table for the largest class of non-ambiguous grammars.

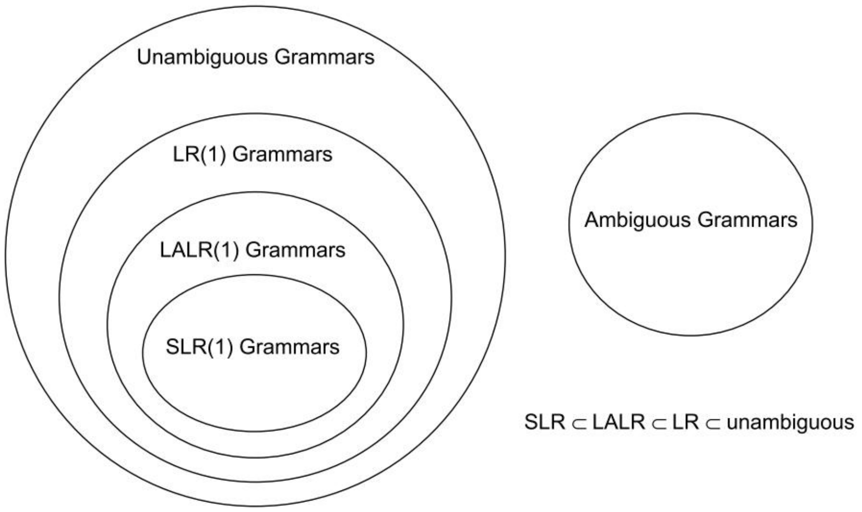


Fig. 3.16 *Applicability of parsing table construction methods on grammars*

3.5.5.7 *Error reporting and recovery in LR parsing* An LR parser can detect an error, when it consults the action table and finds that there is no entry for the given state and input symbol. Errors can never be detected by consulting the goto table. The error detection in LR parsing exhibits valid prefix property—the detection of error happens as soon as the prefix of the input has been seen for which there is no valid continuation for the input scanned till now. A canonical LR parser will not make a single reduction before announcing an error. The SLR and LALR parsers can make several reductions before detecting an error, but will not make a single shift before detecting an error.

The error recovery schemes adopted in LR parsers are:

1. Panic mode recovery.
2. Phrase level recovery.

Consider that the current state is C and the next input symbol is denoted by a. The LR parser on finding that action[current state][a] is an error entry detects an error.

In the panic mode recovery, the LR parser scans down the stack until it finds a state S for which a valid goto[S][A] entry is found, where A is any non-terminal. Now, the input symbols are discarded until an element b in FOLLOW(A) is encountered. The parser then stacks the state goto[S][b] and continues the parsing. The elements in FOLLOW(A) act as the synchronisation token set. Observe that the A can be any non-terminal. There could be multiple choices for A, which needs to be pruned for the particular language.

Phrase-level recovery is implemented by filling in appropriate error handling routines in the action table. These error-handling routines would typically modify the top of the stack or the next input symbol to recover from the error. The caution that the compiler designer has to exercise is to make sure that the error handling routines do not get the parser to go into an infinite loop, due to the modification of stack or input. These error routines are similar to the ones that we saw in operator precedence parsing, but they are easier to write because in LALR parsing, the action table will never have an erroneous reduction, but in operator precedence parsing, there is a chance that we could wrongly reduce. A simple phrase level recovery strategy is as described below. Initially we identify all the expected input symbols in each one of the states. If an input symbol other than the expected ones is encountered in any state, we emit an error message indicating the possible expected inputs. For the states where there is only one particular input symbol α possible, we can have the following error recovery scheme. If the next input symbol is β, we can make state transition to the new state n as if α was the next input symbol. We then verify if there is a valid transition from the new state n for the input symbol β. This verification is necessary to make sure that we do not get into infinite loop. In case there is no valid transition from the new state n for the input symbol β, we cannot recover from the error.

Example 12 shows the phrase-level recovery scheme.

Example 12—Illustration of Error Reporting and Recovery in LR Parsing This example shows the error reporting and a simple phrase level recovery in LR parsing for the parsing table shown in Table 3.23. This is an enhancement made to Example 8, to include the error recovery and reporting.

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-stmt-lex.c c-stmt-lex.l

# Compiling the Lexical Analyzer
$ gcc -c -o c-stmt-lex.o c-stmt-lex.c

# Building ex12 Binary
$ g++ -g -Wall -DCHAP3_EX12 ex12.cc c-stmt-lr-parse.cc c-stmt-lex.o -o ex12

# Variant 7
$ ./ex12 'count=count*2+index;'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 258 (CONSTANT)
Shifting token 258 (CONSTANT), Entering state 2
Reading a token: Next token is 259 (IDENTIFIER)
```

```
Shifting token 259 (IDENTIFIER), Entering state 3
Reading a token: Next token is 260 (OPERATOR)
Reducing via Rule 3
Entering state 5
Shifting token 260 (OPERATOR), Entering state 6
Reading a token: Next token is 257 (EQ_TO_OP)
Shifting token 257 (EQ_TO_OP), Entering state 9
Reading a token: Next token is 260 (OPERATOR)
Reducing via Rule 4
Entering state 5
Shifting token 260 (OPERATOR), Entering state 6
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 8
Reading a token: Next token is 256 (SEMI_COLON)
Reducing via Rule 5
Entering state 5
Shifting token 256 (SEMI_COLON), Entering state 7
Reading a token: Now at end of input.
Reducing via Rule 1
Entering state 10
Success
count=count*2+index;
SYNTAX CORRECT

# Missing Identifier / Constant
$ ./ex12 'count=5+;'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 258 (CONSTANT)
Shifting token 258 (CONSTANT), Entering state 2
Reading a token: Next token is 257 (EQ_TO_OP)
Shifting token 257 (EQ_TO_OP), Entering state 4
Reading a token: Next token is 260 (OPERATOR)
Reducing via Rule 2
Entering state 5
Shifting token 260 (OPERATOR), Entering state 6
Reading a token: Next token is 256 (SEMI_COLON)
Error not finding entry action_table[6][256]..
Expecting IDENTIFIER or CONSTANT
count=5+;
        ^
SYNTAX INCORRECT

# Missing semicolon
$ ./ex12 'count=index'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 258 (CONSTANT)
```

```
Shifting token 258 (CONSTANT), Entering state 2
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 3
Reading a token: Now at end of input.
Error not finding entry action_table[3][0]..
Expecting SEMI_COLON or OPERATOR
count=index
          ^

SYNTAX INCORRECT

# Missing EQTO Operator
$ ./ex12 'count index;'
Entering state 0
Reading a token: Next token is 259 (IDENTIFIER)
Shifting token 259 (IDENTIFIER), Entering state 1
Reading a token: Next token is 259 (IDENTIFIER)
Error not finding entry action_table[1][259]..
Expecting EQ_TO_OP (Inserted it)
Shifting token 259 (IDENTIFIER), Entering state 3
Reading a token: Next token is 256 (SEMI_COLON)
Reducing via Rule 3
Entering state 5
Shifting token 256 (SEMI_COLON), Entering state 7
Reading a token: Now at end of input.
Reducing via Rule 1
Entering state 10
Success
count index;
SYNTAX CORRECT
```

3.6 A SYNTAX ANALYSER FOR C LANGUAGE

In the previous sections, we learnt about the context-free grammar and the techniques used for parsing an input, given the grammar. We also explored how the parser generators use the grammar to automatically generate the parser by using different techniques (like say creating a parsing table from grammar).

This section demonstrates the syntax analyser module of our toy C compiler (mycc) taking in sample C programs and announcing, if the syntax is correct or incorrect.

The syntax analyser module is built from a context-free grammar that describes the syntax of a C language program. We use the 'bison' parser generator program to generate the syntax analyser from the CFG. The generated syntax analyser can take as input, sample C programs and announce, if the syntax is correct or incorrect. This gives an idea on how the specifications of a complete language can be written using a context-free grammar and the corresponding lexical specifications. It illustrates the ease with which the parser can be generated from the CFG. Most of the theory that we have discussed in this chapter goes into making of parser generator programs like 'bison'. This section merely 'uses' the capability of parser generator program to generate a syntax analyser from a context-free grammar describing a C language program.

The following dialog shows the process of making a C language parser from the context-free grammar. It demonstrates how the parser takes different input files and announces if the syntax is conformant to the grammar.

```
# Generating the Parser from Grammar Specifications
$ bison -dy -oc-small-gram.c -v c-small-gram.y

# Compiling the Parser
$ gcc -g -Wall -DGENERATED_PARSER -c -o c-small-gram.o c-small-gram.c

# Generating the Lexical Analyser from lexical Specifications
$ flex -oc-small-lex.c c-small-lex.l

# Compiling the Lexical Analyser
$ gcc -c -o c-small-lex.o c-small-lex.c

# Building the Syntax Analyser Binary
$ gcc -g -Wall c-small-gram.o c-small-lex.o main.c -o syn_analyzer

# Sample C Program with correct syntax
$ cat -n test1.c
     1  /* Function */
     2  int func( int v1,int v2)
     3  {
     4          int v3,v4;
     5
     6          v3=v1+v2;
     7          v4=v1-v2;
     8
     9          return(v3*v4);
    10  }

# Trying out the Syntax Analyser on the Sample Program
$ ./syn_analyzer test1.c

SYNTAX CORRECT

# Sample C Program with syntax error
$ cat -n test1a.c
     1  /* Function */
     2  int func( int v1,int v2)
     3  {
     4          int v3,v4;
     5
     6          v3=v1+v2 /* syntax error - Missing Semicolon */
     7          v4=v1-v2;
     8
     9          return(v3*v4);
    10  }

# Trying out the syntax analyser on the Sample Program
$ ./syn_analyzer test1a.c
```

```
       v4
        ^
Error in Line number=7
SYNTAX INCORRECT

# Sample C Program with syntax error
$ cat -n test1b.c
     1  /* Function */
     2  int func( int v1,int v2)
     3  {
     4          int v3,v4;
     5
     6          v3=v1+v2;
     7          v4=v1 - / v2; /* syntax error - missing operand */
     8
     9          return(v3*v4);
    10  }

# Trying out the syntax analyser on the Sample Program
$ ./syn_analyser test1b.c

       v4=v1 - /
               ^
Error in Line number=7
SYNTAX INCORRECT

# Sample C Program with no syntax error,but semantic error
$ cat -n test1c.c
     1  /* Function */
     2  int func( int v1,int v2)
     3  {
     4          int v3,v4;
     5
     6          v3=v1+v2;
     7          v4=v1-v2;
     8
     9          /*
    10          Undefined variable 'v5' used below in return statement
    11          It is a Semantic Error.
    12          However it is NOT a syntax error
    13          */
    14          return(v5*v4);
    15  }

# Trying out the Syntax Analyser on the Sample Program
$ ./syn_analyzer test1c.c

SYNTAX CORRECT
```

From the different inputs given to the toy parser discussed so far, there are a couple of observations that can be made.

The toy parser can check the syntax of an input C program and announce if the syntax is correct or erroneous. In case, there is a syntax error, it reports the line number in which the error is found.

The input 'test1c.c' contains the use of an undeclared variable 'v5' in the return statement. However, the syntax analyser did not pronounce any syntax error. This is because, from a grammar point of view, there is no syntax error. The sequence of tokens in the statement conforms to the grammar specifications. However, such errors are caught in 'semantic analysis', which is the topic of discussion in the next chapter. This sample input file illustrates the important distinction between syntax and semantic analysis.

SUMMARY

Syntax analysis is the second step in compilation process of a source program after the lexical analysis. In lexical analysis phase, the input source program is broken up into a sequence of tokens. The syntax analysis (also called as parsing) verifies if the sequence of tokens returned by the lexical analyser are in accordance to the grammar of the language. A notation known as context-free grammar (CFG) is used to describe the grammar of a language. There are various techniques used for parsing an input source program to verify if the tokens are in accordance to the grammar described by CFG. The techniques can be classified as top-down parsing or bottom-up parsing. We studied about the top-down parsing in Section 3.4 and bottom-up parsing in 3.5. We learnt about the two major methods of top-down parsing, namely the recursive descent parsing and the table-driven predictive parsing. In bottom-up parsing, we studied about operator precedence parsing and the LR parsing. The LR parsing technique is the most powerful one since it can be applied to a large variety of grammars. In LR parsing, a couple of tables called as the action table and the goto table are used to determine if the tokens from the input source program are in accordance to the specified grammar. There are 3 methods of deriving these tables automatically from the grammar of the language. They are called as simple LR (SLR), canonical LR and look ahead LR methods of parse table generation. The look ahead LR method of constructing the parsing tables is used in most of the popular parser generators like 'bison' and 'yacc'.

One of the responsibilities of a syntax analyser is to detect errors in the input source program and report them to the programmer to take of correcting it. This is called as error reporting. The syntax analyser cannot stop at the first error encountered in the input source program, it should continue and detect as many errors as present in a source program as possible. This is called as the error recovery in a syntax analyser. Section 3.3 discusses various strategies to recover from errors and continue parsing. During the discussion of individual parsing strategies, we also examined the error recovery schemes that can be used in tandem with them.

REVIEW QUESTIONS AND EXERCISES

3.1 What is syntax analysis? What are its primary functions? What are its secondary functions?

3.2 State whether the following statements are true or false.

(a) A syntax analyser verifies if the tokens for a given input are properly sequenced in accordance with the grammar of the language under check.
(b) A syntax analyser detects the error, emits appropriate error message to the user and if possible, recovers from the error for a given input.
(c) A syntax analyser reads an input source program and produces as output a sequence of tokens.
(d) A syntax analyser generates intermediate code.

3.3 What is a context-free grammar? Illustrate with an example the different components of a context-free grammar. What are the advantages of using context-free grammar to specify a language?

3.4 How do you prove that an input is syntactically in conformance to grammar? Provide the proof that the input '***min= count - index +5;***' is conformant to the grammar

c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression → CONSTANT
c_expression → IDENTIFIER
c_expression → c_expression OPERATOR c_expression

Assume that the lexical analyser is capable of splitting the input into tokens IDENTIFIER, EQ_TO_OP, SEMI_COLON, CONSTANT and OPERATOR. Also assume that the lexical analyser returns OPERATOR as a token for any of the operators like + or –.

3.5 What is a derivation? Illustrate with an example, the leftmost derivation and rightmost derivation.

3.6 What is a parse tree? Show the generated parse tree for an input '***m = n + p –q;***' while checking for conformance with the grammar given in Q. 3.4. From the parse tree, identify each of the replacement of the productions in the derivation.

3.7 Give an example of an ambiguous grammar. What are the techniques to disambiguate grammars?

3.8 How do you classify the different parsing techniques?

3.9 Distinguish between error reporting and error recovery in a parser. What are the main considerations for each of them in a parser?

3.10 List out the important error recovery strategies in a parser.

3.11 Distinguish between top-down parsing and bottom-up parsing? What is the largest class of grammars that can be parsed by each of them?

3.12 Illustrate with an example the working of a backtracking parser? List out its advantages and disadvantages.

3.13 What is a predictive parser? What is the need to modify a grammar to suit predictive parser? Illustrate with examples.

3.14 What is a recursive descent parser? How do you implement a recursive descent parser for a grammar?

c_statement → IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression → CONSTANT c_expr_rest
c_expression → IDENTIFIER c_expr_rest
c_expr_rest → OPERATOR c_expr_factor
c_expr_rest → ϵ
c_expr_factor → IDENTIFIER c_expr_rest
c_expr_factor → CONSTANT c_expr_rest

3.15 What are the components of a table-driven predictive parser? Which of those are dependent on the grammar that is being parsed? What are the advantages of a table-driven predictive parser?

3.16 Illustrate with an example the moves of a table-driven predictive parser for a given input and the context-free grammar. You can use the parsing table corresponding to the grammar for helping the moves of the table-driven predictive parser.

3.17 Illustrate by example, how a bottom-up parser reduces a given input to the start symbol. Draw the parse tree for the same and prove that the reduction is the exact reverse of a rightmost derivation.

3.18 What transformations make a grammar suitable for bottom-up parsing? Illustrate with an example.

3.19 Define the terms reduction, handle and right sentential form. Explain with an example, the importance of picking the right-handles during a reduction sequence.

3.20 Illustrate with example, the working of a general bottom-up parser using the shift and reduce actions.

3.21 Develop a bottom-up parser for the grammar given below using 'bison', the parser generator program.

c_statement $\rightarrow$ IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
c_expression $\rightarrow$ CONSTANT
c_expression $\rightarrow$ IDENTIFIER
c_expression $\rightarrow$ c_expression OPERATOR CONSTANT
c_expression $\rightarrow$ c_expression OPERATOR IDENTIFIER

Check the parser for the inputs marked as variant 1 through variant 7 in Section 3.2.

3.22 How does an operator precedence parser work? Use a pre-constructed operator precedence table to guide the parsing of an input '***a + b – 20***' using operator precedence parser.

3.23 Describe the error reporting and recovery schemes in operator precedence parsing?

3.24 What are precedence functions in an operator precedence parser? List out the advantages and disadvantages of operator precedence parsing.

3.25 What are the main components of an LR parser? Describe them.

3.26 Illustrate the steps in the parsing of an input 'x = y + z – 5;' by an LR parser using a pre-constructed LR parsing table.

3.27 What are the different methods of constructing the LR parsing table from a grammar? What are the common steps involved in constructing the LR parsing table from a given grammar?

3.28 What is an LR(0) Item? How are closure and goto operations performed on LR(0) Item set?

3.29 Describe the SLR(1) method of constructing the LR parsing table from a given grammar. Illustrate with an example.

3.30 Describe the canonical LR method of constructing the LR parsing table from a given grammar. Illustrate with an example.

3.31 Describe the look ahead LR method of constructing the LR parsing table from a given grammar. Illustrate with an example.

3.32 How do SLR(1), LR(1) and LALR(1) methods compare against each other in the process of constructing the parsing table from the grammar? Draw a set diagram depicting the applicability of SLR(1), LR(1) and LALR(1) methods on the grammars.

3.33 How does the error detection and error recovery happen in LR parsing? Illustrate with an example.

3.34 Add error detection and recovery capabilities to the LR parser developed in 3.21.

3.35 Distinguish between a syntax error and a semantic error by providing an example.

SEMANTIC ANALYSIS

4

Introduction

In the previous chapters we had studied about how in a compiler, the input is broken up into tokens (lexical analysis) and how the token ordering is checked to see if it is conformant to the grammar of the language (syntax analysis). All the statements that are valid from syntax point of view are not legal instructions. For example, a statement in a C language program *x* = *y*( ) is valid from syntax point of view. However, it is not a legal statement, if y is defined as an integer instead of being defined as a function. In semantic analysis we verify if the input source forms a legal set of instructions in accordance with the language rules. For example, semantic analysis in a C language compiler would involve tasks like (a) verifying if all the variables are declared before use; (b) checking if expressions and variables are being used properly keeping the data types in mind, e.g. LHS and RHS in an assignment statement should have the same data type; (c) checking if the operators are used on compatible operands, e.g. the operator '->' is used on a variable, which is a pointer to structure, the operator '*' is used on pointer variable, and so on. Semantic analysis also detects errors like defining an identifier more than once in the same scope. In the case of object-oriented languages, semantic analysis additionally detects issues like violation of access controls, and so on. Semantic analysis also involves gathering information that would be used in later phases like intermediate and target code generation. The semantic analysis is the last phase in which we reject incorrect input programs and flash error messages for the user to correct them.

The following dialog examines a few C programs, which have some semantic errors and shows us how the GNU C compiler detects and reports them. These examples give us a feel of what kinds of errors are detected in semantic analysis. Observe that all of these programs are syntactically correct, but have semantic errors.

```
# A C Program using an undeclared variable
$ cat -n sem_err1.c
     1
     2   int main()
     3   {
     4        int a,b;
     5
     6        a=1;
     7        b=2;
     8        c=3; /* Use of undeclared variable */
     9
    10        a = b + c;
    11
    12        return(a);
    13
    14   }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err1.c -o sem_err1
sem_err1.c: In function `main':
sem_err1.c:8: error: `c' undeclared (first use in this function)
sem_err1.c:8: error: (Each undeclared identifier is reported only once
sem_err1.c:8: error: for each function it appears in.)

# A C Program Assigning a float to char pointer
$ cat -n sem_err2.c
     1
     2   int main()
     3   {
     4        char *a;
     5
     6        float b,c;
     7
     8        b = 30.45;
     9        c = 40.36;
    10
    11        a = b + c; /* Assigning a float to char pointer */
    12
    13        return(0);
    14
    15   }
```

```
# The Compiler detects it and reports the error
$ gcc -Wall sem_err2.c -o sem_err2
sem_err2.c: In function `main':
sem_err2.c:11: error: incompatible types in assignment

# A C Program using '->' operator on a float Variable
$ cat -n sem_err3.c
     1
     2  int main()
     3  {
     4
     5      float b,c;
     6
     7      b = 30.45;
     8      c = 40.36;
     9
    10      b = c -> f1; /* using '->' operator on a float Variable */
    11
    12      return(0);
    13  }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err3.c -o sem_err3
sem_err3.c: In function `main':
sem_err3.c:10: error: invalid type argument of `->'

# A C Program using break statement in a non-loop context
$ cat -n sem_err4.c
     1
     2   int main()
     3   {
     4        float b,c;
     5
     6        b = 30.45;
     7        c = 40.36;
     8
     9        b = c ;
    10
    11        break; /* using break statement in a non-loop context */
    12
    13        return(0);
    14   }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err4.c -o sem_err4
sem_err4.c: In function `main':
sem_err4.c:11: error: break statement not within loop or switch

# A C Program using a float variable as function
$ cat -n sem_err5.c
     1
     2   int main()
     3   {
     4        float b,c;
     5
     6        b = 30.45;
     7        c = 40.36;
     8
     9        b = c() ; /* using a float variable as function */
    10
    11        return(0);
    12   }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err5.c -o sem_err5
sem_err5.c: In function `main':
sem_err5.c:9: error: called object is not a function

# A C Program declaring a variable twice in the same scope
$ cat -n sem_err6.c
     1
     2   int main()
     3   {
     4        int a;
     5
     6        int b,c;
     7        float a; /* declaring a variable twice in the same scope */
     8
     9        b = 30;
    10        c = 40;
    11
    12        a = b + c;
    13
    14        return(a);
    15
    16   }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err6.c -o sem_err6
sem_err6.c: In function `main':
sem_err6.c:7: error: conflicting types for 'a'
sem_err6.c:4: error: previous declaration of 'a' was here
sem_err6.c:4: warning: unused variable 'a'
```

Semantic analysis involves the following main tasks:

1. Process the declarations (e.g. variable declarations, function prototype declarations, type definitions, etc.) and build/update the symbol table to record the type information associated with various Identifiers.
2. Examine the rest of the program to ensure that the identifiers are used correctly adhering to the type-compatibility conventions defined by the language. For example, one of the type-compatibility conventions in C language is that the LHS and RHS in an assignment statement

should match. Another one of the type-compatibility conventions in C language is that the variable names used should be unique in that particular scope. Each programming language has its own set of type-compatibility conventions that must be met for a program to be declared as semantically correct. This assessment of program for type-compatibility is called ***type checking***.

We had used regular expressions to specify and implement the lexical analysis. For the syntax analysis, we had used the context-free grammar to specify and implement the parsers. In order to perform semantic analysis, we make use of a formalism called ***syntax directed translation (SDT)***. The syntax-directed translation allows us to specify and implement a semantic analyser. We can see in the next chapter how the syntax-directed translation technique can also be used for generation of intermediate code. We study the details of syntax-directed translation in Section 4.1. We show the application of the SDT technique to perform the semantic analysis in Section 4.2. This is followed by a sample semantic analyser implementation in Section 4.3.

4.1 SYNTAX-DIRECTED TRANSLATION

The main idea behind syntax-directed translation is that the semantics or the meaning of the program is closely tied to its syntax. Most of the modern compiled languages exhibit this property.

Syntax-directed translation involves:

- Identifying attributes of the grammar symbols in the context-free grammar.
- Specifying ***semantic rules*** or attribute equations relating the attributes and associate them with the productions.
- Evaluating semantic rules to cause valuable side-effects like insertion of information into the symbol table, semantic checking, issuing of a error message, generation of intermediate code, and so on.

An attribute is any property of a symbol. For example, the data type of a variable is an attribute. The memory location associated with a variable is another attribute. In the next few sections, we shall see many attributes of the grammar symbols in use.

The syntax-directed definition is a commonly used notation for specifying attributes and semantic rules along with the context-free grammar.

4.1.1 Syntax Directed Definition

Consider the context-free grammar for recognising declarations in C language supporting the basic data types like 'int', 'char' and 'float', shown in Table 4.1.

Table 4.1 *Grammar for supporting C-declarations*

#	Production		
1	declaration_list	→	declaration_list declaration
2		|	declaration
3	declaration	→	type_spec identifier_list ';'
4	type_spec	→	INT
5		|	CHAR
6		|	FLOAT
7	identifier_list	→	identifier_list ',' IDENTIFIER
8		|	IDENTIFIER

A sample input source that conforms to the grammar of Table 4.1 is shown in Table 4.2.

Table 4.2 *An input source*

```
int a,b;
```

Let's examine the grammar shown in Table 4.1 and get to understand some of the definitions and concepts.

- The syntactic entities (terminals and non-terminals) in the grammar shown in Table 4.1 are declaration_list, declaration, type_spec, identifier_list, INT, CHAR, FLOAT and IDENTIFIER. The start symbol is 'declaration_list'. The terminals INT, CHAR, FLOAT are tokens returned by lexical analyser for the keywords 'int', char' and 'float' respectively.
- There are ***attributes*** that can be defined for the above syntactic entities. For example, the syntactic entity-identifier_list can have an attribute 'type', which is representative of the data type of all the identifiers in the list.
- The attribute for a grammar symbol is referred to by writing the grammar symbol followed by a dot and the attribute name. For example, the 'type' attribute of identifier_list can be referred to as identifier_list.type. Observe that this notation is similar to accessing a field in a structure in C language.
- The value of an attribute can be a string or a number or any other convenient value. For example, the 'type' attribute of the identifier_list (identifier_list.type) can have one of the value from the set {INTEGER, CHARACTER, REAL}.
- The attributes of different syntactic entities are related using ***semantic rules*** associated with each production. For example, we can associate a semantic rule "identifier_list.type= type_spec. data_type" with the Production 3. This signifies that the value of 'type' attribute of identifier_list is copied from the data_type attribute of the type_spec in association with Production 3.
- A context-free grammar in which the productions are shown along with its associated semantic rules is called as a ***syntax-directed definition***. Table 4.3 shows the syntax-directed definition evolved from the grammar for the C declarations seen in Table 4.1.

Table 4.3 *Syntax-directed definition*

	Production			Semantic Rule
1	declaration_list	→	declaration_list declaration	
2		|	declaration	
3	declaration	→	type_spec identifier_list ';'	identifier_list .type = type_spec. data_type
4	type_spec	→	INT	type_spec.data_type = INTEGER
5		|	CHAR	type_spec.data_type = CHARACTER
6		|	FLOAT	type_spec.data_type = REAL
7	identifier_list	→	identifier_list ',' IDENTIFIER	identifier_list$_1$.type = identifier_list.type insert (IDENTIFIER.place, identifier_list.type)
8		|	IDENTIFIER	insert (IDENTIFIER.place, identifier_list.type)

- In a syntax-directed definition, while writing the semantic rules for a production, if the same grammar symbol appears on both the left and right side of the production, then each occurrence on the right-hand side is referred to by a subscript. For example, the production 7 in Table 4.3, shows a semantic rule 'identifier_list$_1$.type = identifier_list.type'. The identifier_list$_1$.type refers to the attribute 'type' of the grammar symbol 'identifier_list' on the right-hand side of the production, while the identifier_list. type (without subscript) refers to the 'type' attribute of 'identifier_list' on the left-hand side of the production.

- The semantic rules in a syntax-directed definition can also be used to call routines that can provide side-effects like generation of code, insertion/updating of information into symbol table, perform a semantic check, issue warning messages, and so on. For example, the semantic rule associated with Production 7 in Table 4.3, we call a routine 'insert', which inserts a symbol table entry for the IDENTIFIER with the type information, given by the attribute identifier_list.type.

Consider the source shown in Table 4.2 as an input to the syntax-directed definition presented in Table 4.3. Figure 4.1 illustrates the parse tree showing the values of attributes of the parse tree nodes for the input. A parse tree showing the values of attributes of different parse tree nodes is called a ***decorated parse tree***. The process of computing the attribute values at various parse tree nodes is called decorating or annotating the parse tree.

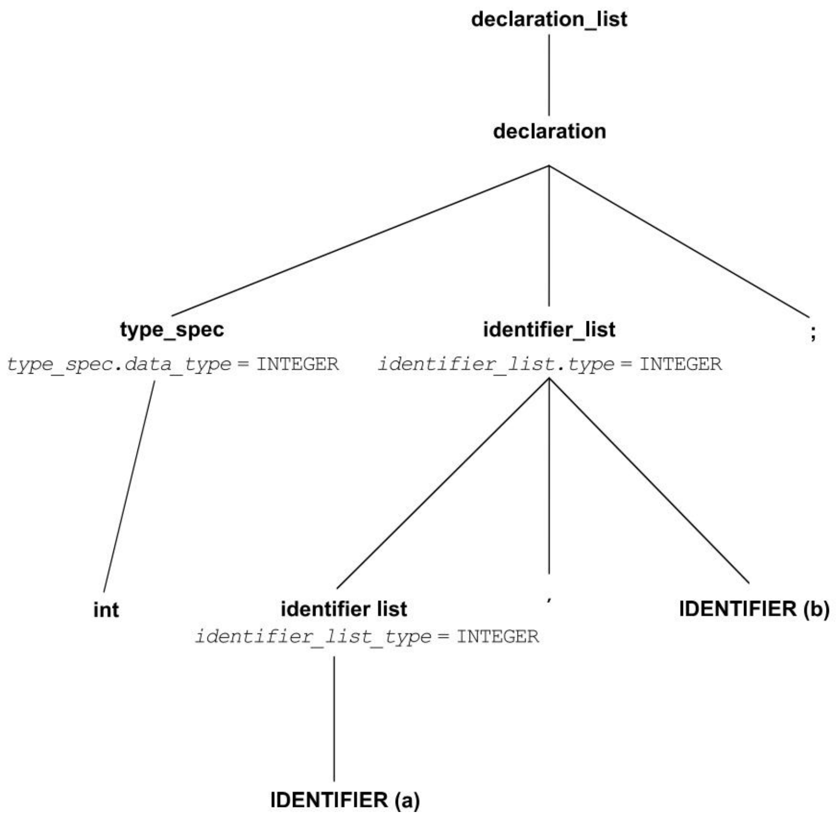


Fig. 4.1 *Decorated parse tree*

A parse tree might be decorated during the parsing process itself. In other cases, the base parse tree could be constructed during the parsing. After the parse tree is constructed, the attributes could be evaluated and parse tree decorated by visiting the tree nodes. The order in which the parse tree nodes are visited for evaluating the attributes is based on the dependencies of the attributes belonging to various nodes. For example, the semantic rule for the Production 3 is identifier_list.type = type_spec.data_type, i.e. the type attribute of identifier_list is assigned the value of data_type attribute of type_spec node. This tells us that in order to compute the attribute 'type' of identifier_list, the attribute 'data_type' for the type_spec node needs to have already been computed. Similarly, for Production 7, in order to compute identifier_list$_1$.type, the attribute identifier_list.type should have already been computed. Thus, in order to properly decorate the above parse tree, we need to compute the 'data_type' attribute of type_spec node first, followed by 'type' attribute of identifier_list, and so on. This dependency of attributes of a parse tree node on other attributes is illustrated by a ***dependency graph***. Figure 4.2 shows the dependency graph overlaid on the parse tree. In a dependency graph, the direction of the arrow signifies the order in which the attributes are evaluated. The parse tree is shown in the dotted line in the background.

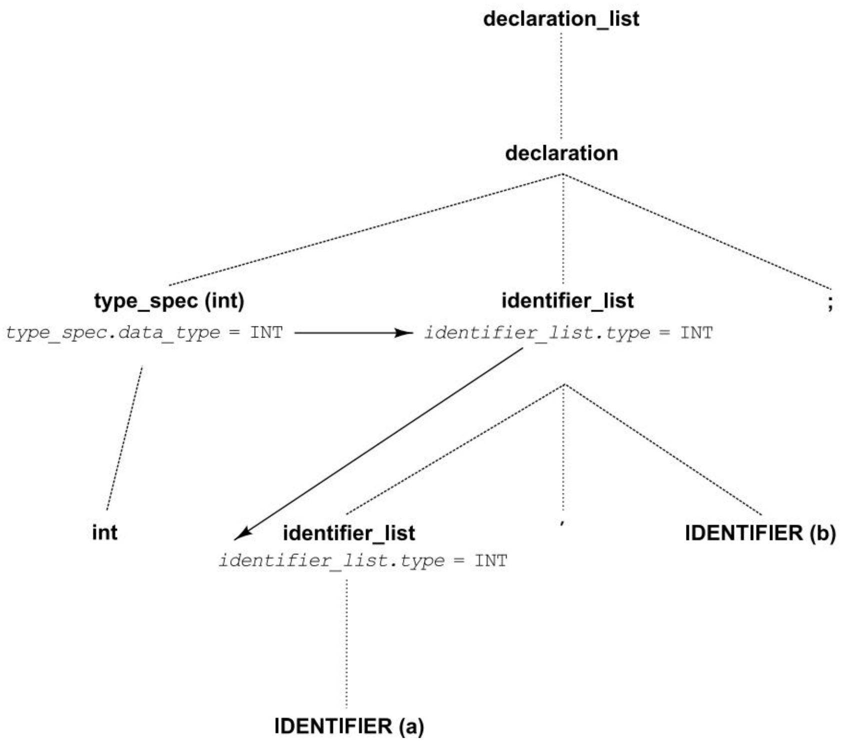


Fig. 4.2 *Dependency graph*

Let's briefly understand the nature of the attributes in a syntax-directed definition. Consider the syntax-directed definition shown in Table 4.3. In the semantic rule for Production 7, we can observe that the attribute of the element on the RHS of the production (identifier_list$_1$.type) is dependent on attribute of the element in the LHS of the production (identifier_list.type). In the parse tree, the symbol on the LHS of the production is a parent and the symbols on the RHS of the productions are the children as seen in Fig. 4.2. In other words, the attribute identifier_list$_1$.type depends on its parent attribute – identifier_list. type. Such attributes for a grammar symbol that depend on its parent's or a sibling's attribute are called ***inherited attributes***. In Production 3 of the same SDD, we see another example of an inherited attribute—identifier_list.type, which depends on its sibling's attribute—type_spec. data_type.

It is also possible to have an attribute of a grammar symbol that depends on one or more of its children's attributes. Such attributes are called as ***synthesised attributes***. A syntax-directed definition that makes use of synthesised attributes only is known as an ***S-attributed definition***. Table 4.4 shows an S-attributed syntax-directed definition.

Table 4.4 *An S-attributed syntax-directed definition*

#	Production			Semantic Rule
1	sentence	:	expr	print expr.value
2	expr	:	expr '+' term	expr.value = expr_1.value + term.value
3		|	expr '-' term	expr.value = expr_1.value - term.value
4		|	term	expr.value = term.value
5	term	:	term '*' factor	term.value = term_1.value * factor.value
6		|	term '/' factor	term.value = term_1.value / factor.value
7		|	factor	term.value = factor.value
8	factor	:	'(' expr ')'	factor.value = expr.value
9		|	CONSTANT	factor.value = CONSTANT.lexeme

The syntax-directed definition in Table 4.4 describes a desktop calculator taking expressions involving constants as input and printing out the result of the expression as output. Table 4.5 shows some sample input and the corresponding output from the SDD.

Table 4.5 *Sample input and output*

Input	Output
9+15–20	4
3*21 – (4*5)	43
(9*53) /(7–4)	159

4.1.2 Evaluation of Semantic Rules in SDD

In the last section, we studied about how a syntax-directed definition can be used for specifying attributes and semantic rules along with the context-free grammar. In this section, we look at the methods by which the semantic rules of an SDD can be evaluated to compute the values of attributes. The evaluation of semantic rules in a syntax-directed definition achieves the actual translation of input to the output.

One of the methods for computing the attributes of different symbols in a syntax-directed definition is to evaluate the semantic rules during the parsing itself. This method of ***translation interleaved with parsing*** can be applied to a restricted class of syntax-directed definitions.

Another method of evaluating semantic rules is to construct a parse tree, form a dependency graph and then evaluate the semantic rules in accordance with the dependency graph. This method of evaluation is called the ***parse tree method***. This method works for any kind of SDD provided there is no cycle in the dependency graph. The parse tree method takes more time for evaluating the semantic rules, since it involves multiple passes of the input (one for creating parse tree, and one or more for evaluating the semantic rules).

Another method of evaluating the semantic rules is called as the ***rule-based method***. In the rule-based method, a parse tree is created from the syntax analysis similar to the parse tree method, but the evaluation order is determined beforehand by analysis of the semantic rules. The dependency graph is not created in this method. In the rule based method, the order of evaluation of attributes is hard-coded into the compiler. In order to completely evaluate all the semantic rules, it is possible that the parse tree is walked through multiple times in the rule-based method.

We study in detail about each of the above-mentioned methods to evaluate semantic rules of a SDD in the next few sections.

4.1.3 Translation Interleaved with Parsing

Semantic rules can be evaluated in a single pass during parsing itself, without having to explicitly create a parse tree or a dependency graph. The evaluation of semantic rules during the parsing is applicable to a class of syntax-directed definitions whose semantic rules meet certain criteria. The order of evaluation of attributes for the syntax-directed definition plays an important part in determining the feasibility of evaluation of semantic rules during the parsing.

Even if there is no explicit parse tree that is created, an order of evaluation of attributes that is a reference for evaluation of semantic rules during the parsing is the ***depth-first order*** of evaluation of attributes. In depth-first traversal, an entire sub-tree under each child is traversed before going to the next child. Figure 4.3 shows a parse tree and the order in which the nodes are visited by depth-first traversal.

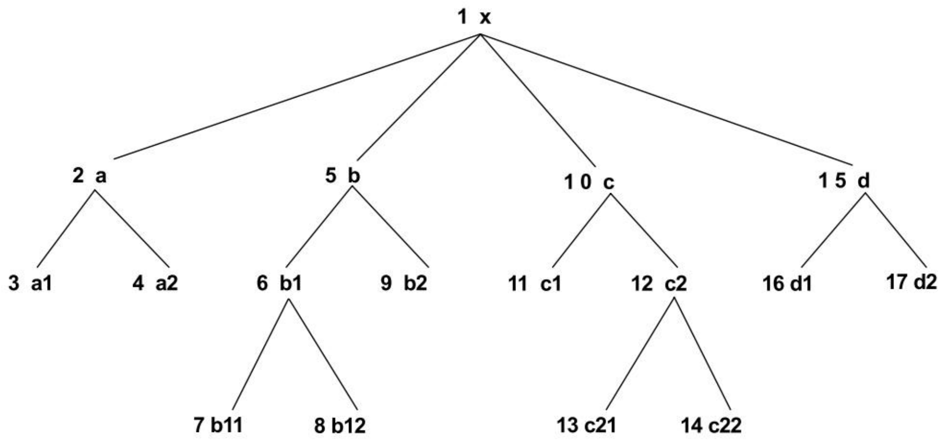


Fig. 4.3 *Depth-first traversal of parse tree*

The depth-first order of evaluation of attributes can be represented by Algorithm 4.1.

```
procedure dfeval ( node *n)
{
      for each of the child m of n from left to right
      {
            Evaluate inherited attributes of m
            dfeval (m);
      }

      /* At this point all the children of Node n have been processed */
      Evaluate synthesised attributes of Node n.
}
```

Algorithm 4.1 *Depth-first traversal for evaluation of attributes*

Consider a production A → X1 X2 X3 X4 ..Xj … Xn. in a context-free grammar. For the successful evaluation of attributes using depth-first evaluation method, we can see that the attributes of nodes have to adhere to the following guidelines.

1. The inherited attributes of any particular node Xj can depend on the attributes of X1,X2…Xj–1 but not on any of the right siblings like Xj+1 ..Xn. This is because the attributes of right siblings of Xj like Xj+1... Xn would not yet have been evaluated by the depth-first traversal algorithm at the time of evaluating Xj.
2. The inherited attributes of any particular node Xj can depend on A. This is because the node A would have already been evaluated before evaluating Xj.

The syntax-directed definitions that adhere to the above guidelines and lend themselves to depth-first evaluation of attributes are known as ***L-attributed definitions***. It is useful to observe that all the S-attributed definitions are also L-attributed definitions because the guidelines (1) and (2) apply only for inherited attributes. The syntax-directed definition presented in Table 4.3 is L-attributed, since none of the productions violate the above two guidelines. The semantic rules for L-attributed definitions can be evaluated during the parsing stage itself.

4.1.3.1 *Translation Scheme*

A translation scheme is a useful notation for specifying ***translation during parsing***. A translation scheme allows us to specify the semantic rules expressing the relationship between attributes as well as the order of evaluation of attributes. A translation scheme is a context-free grammar in which the attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions. The position at which the action can be executed is shown by enclosing it in a bracket and inserting it in the right-side of the production. The semantic actions can refer to inherited as well as synthesised attributes. A translation scheme that recognises variable declarations in C language and makes entries into the symbol table is shown in Table 4.6. This translation scheme is a manifestation of the SDD we studied earlier in Table 4.3.

Table 4.6 *Translation Scheme*

#	Production		
1	declaration_list	:	declaration_list declaration
2		|	declaration
3	declaration	:	type_spec { identifier_list.type = type_spec.data_type } identifier_list ‘ ;’
4	type_spec	:	INT { type_spec.data_type = INT }
5		|	CHAR { type_spec.data_type = CHAR }

6		|	FLOAT { type_spec.data_type = FLOAT }
7	identifier_list	:	{ identifier_list$_1$.type = identifier_list.data_type } identifier_list ‘,’ IDENTIFIER { insert(IDENTIFIER.place,identifier_list.type) }
8	identifier_list	:	IDENTIFIER { insert (IDENTIFIER.place, identifier_list.type) }

When drawing a parse tree for a translation scheme, we indicate a semantic action by constructing an extra child for it, connected by a dashed line to the node of production. Since there are no children for the semantic action, it gets executed when it is first seen during the ***depth-first traversal***. The parse tree might not be explicitly created during a translation. It merely serves as a way of expressing the order of evaluation of attributes in a translation scheme.

The parse tree for the simple input source as shown in Table 4.2 parsed using the translation scheme in Table 4.6 is shown in Fig. 4.4. The depth-first traversal of the parse tree showing the dotted semantic actions gives the order of evaluation of the attributes in a translation scheme. The order of evaluation of attributes in Fig. 4.4, which is arrived by depth-first traversal of the tree, is marked sequentially from 1 through 5.

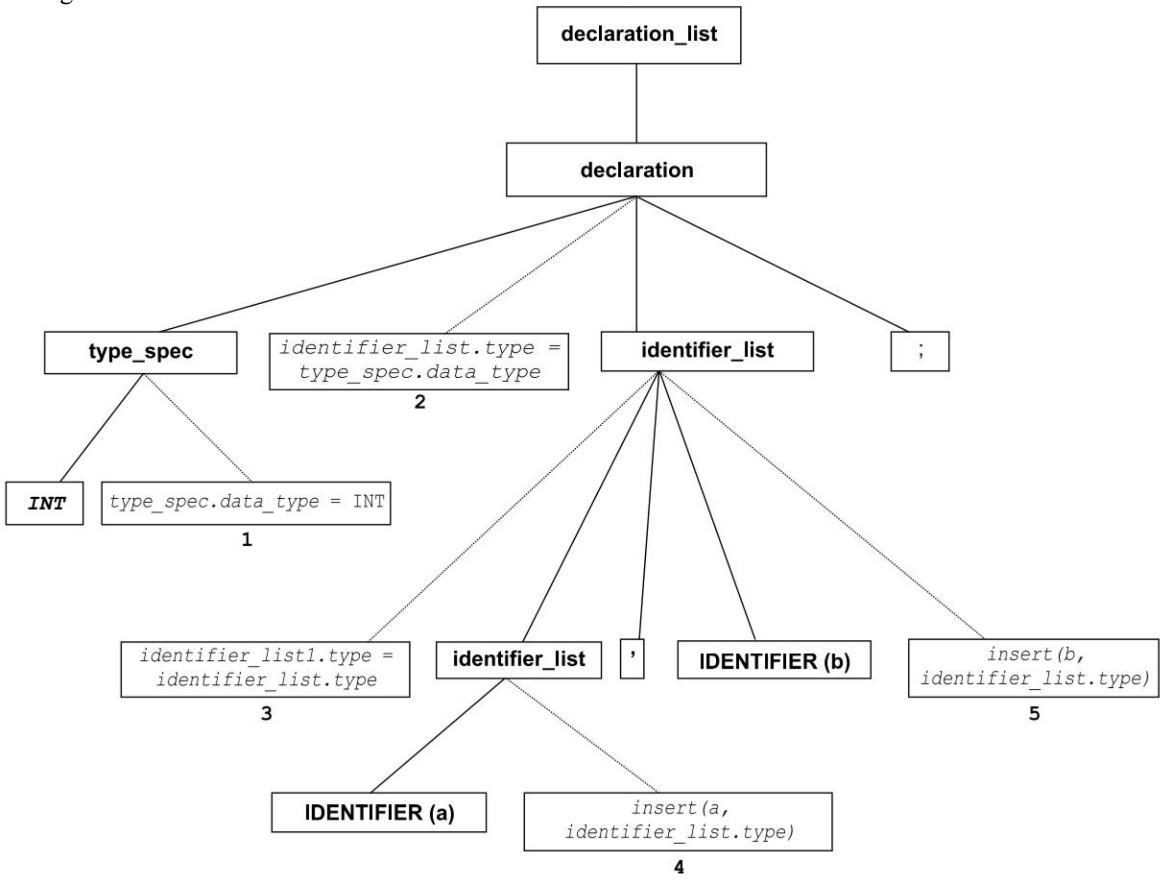

declaration_list
declaration
type_spec
identifier_list.type = type_spec.data_type
2
identifier_list
;
INT
type_spec.data_type = INT
1
identifier_list1.type = identifier_list.type
3
identifier_list
,
IDENTIFIER (b)
insert(b, identifier_list.type)
5
IDENTIFIER (a)
insert(a, identifier_list.type)
4

Fig. 4.4 *Semantic actions shown in parse tree*

Similar to the syntax-directed definition notation, the semantic actions in a translation scheme can compute the value of attributes or call routines to produce side-effects like printing a value or inserting/

updating symbol table entry. In Fig. 4.4 we can see the semantic actions marked 4 and 5 insert an entry in the symbol table.

Translation schemes are designed to ensure that the semantic actions refer to attributes whose values are already computed by a previous semantic action or by the lexical analyser. Note that this is a significant departure from the convention that we followed in SDD specifications. In SDD, we merely specify how the value of certain attribute ‘x’ of a symbol A is computed from attribute ‘y’ of another symbol B. There is no guarantee that the attribute ‘y’ of symbol B is already computed. This is the reason why we had to look at dependency graph and evaluation order for evaluating semantic rules, while dealing with a SDD. In the translation scheme, the design ensures that we refer to attributes that have already been computed by previous semantic actions. It is always possible to construct a translation scheme from an L-attributed syntax-directed definition.

Since the translation scheme needs to ensure that the semantic actions refer to attributes whose values are already computed by a previous semantic action, we need to adhere to the following rules while designing it:

- An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol.
- An action must not refer to a synthesised attribute of a symbol to the right of the action.
- A synthesised attribute for the non-terminal on the left can be only computed after all attributes it references have been computed. The action computing such attributes are usually placed at the end of the right-side of the production.

It is important to understand that the above guidelines essentially stem out of the fact that the semantic actions in a translation scheme are executed in the order of depth first traversal of the parse tree (even if the parse tree is not explicitly created).

In the next few sections and also in the forthcoming chapters, we make use of translation schemes for performing semantic analysis, generation of intermediate code, and so on.

4.1.3.2 *Bottom-Up Translation* The evaluation of semantic rules during the bottom-up parsing of the input source is called as ***bottom-up translation***. In this section, we look at techniques to evaluate semantic rules during the bottom-up parsing for L-attributed definitions.

Let’s have a quick recap of the bottom-up parsing before we get into translation interleaved with it. The most common bottom-up parsing method is the LR parsing method, which was explained in Section 3.5.5. An LR parser consists of an input, an output, a state stack, driver program and a parsing table made up two parts (action and goto). Algorithm 3.4 explained how the LR parsing of the input proceeds. There are 2 steps involved in bottom-up parsing given the input source and the grammar for which we are checking the compliance. The first step is to create a parsing table (action and goto tables) from the grammar. The second step is to parse the input source as explained in Algorithm 3.4, using the parsing table. The LR parsing process allows us to execute fragments of code when a reduction takes place. This facility of executing the code fragments during a reduction is used for evaluating semantic rules. The evaluation of semantic rules allows us to perform semantic analysis, generation of intermediate code and other useful side-effects.

In order to facilitate the evaluation of semantic rules and translation of input source an additional stack containing the ***values of the attributes*** of symbols involved in the reduction is used. This stack (called **VAL stack**) supplements the state stack, which is primarily used for making parsing decisions of shift/reduce. The shift action would push the value of an attribute corresponding to the input grammar symbol on the VAL stack. A reduce action would pop as many elements as present in the RHS of the production and push the attribute’s value of LHS of the production on the VAL stack. Figure 4.5 shows the VAL stack before and after a shift or a reduce operation. In cases where the symbol to be shifted does not have any attribute (e.g. operators like +, –, *), the value that is pushed on VAL stack is undefined. In cases, where there are multiple attributes to a grammar symbol, a pointer to a structure containing the attributes can be pushed on to a VAL stack on a shift operation. The same can be popped off the VAL stack during a reduce operation.

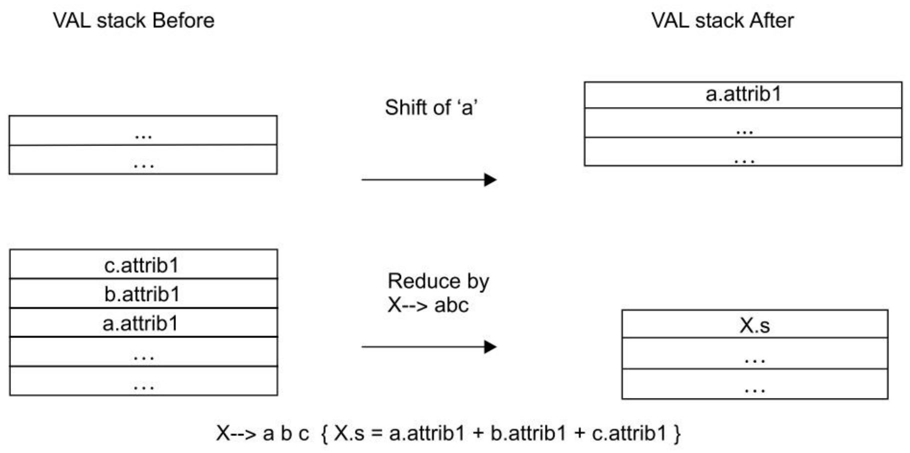


Fig. 4.5 *VAL stack before and after shift/reduce operations*

The LR parser generators like 'bison'/ 'yacc,' etc. allow for maintaining and accessing the VAL stack by means of special $ variables. The special variables $1, $2, etc. correspond to the VAL stack for each element in the RHS of the production. The attribute of the non-terminal on the LHS of the production can be created using the special variable $$. For example, during the reduction of a production x → a b c, the attribute of 'a' on the RHS is accessible using $1, the attribute of 'b' is accessible using $2 and the attribute of 'c' is accessible using $3. The attributes of the LHS non-terminal 'x' can be created using the special variable $$.

The VAL stack helps us evaluate the synthesised and inherited attributes during the bottom-up parsing of an L-attributed definition.

Consider the production x → a b c { x.s = a.attrib1+ b.attrib1 + c.attrib1 }. For an input conformant to the grammar, the VAL stack just before the reduction of the production x → a b c would be as shown in Fig. 4.6. The VAL stack contains the values of attributes of the children a.attrib1, b.attrib1 and c.attrib1. Using these values, we can compute x.s, a synthesised attribute of 'x' and store it on the VAL stack. This shows how we can evaluate synthesised attributes of symbols using the VAL stack during the bottom-up translation.

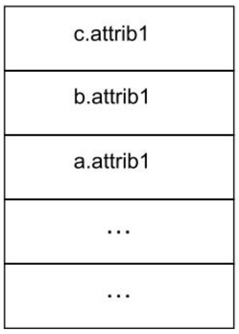


Fig. 4.6 *VAL stack contains attributes of children before reduction*

Consider the syntax directed definition in Table 4.7 that has semantic actions involving inherited attribute depending on the synthesised attributes of its left-side siblings in a production. We can see from

Table 4.7 that the inherited attribute d.i depends on the synthesised attributes of left-side siblings b.s and c.s of Production 1.

Table 4.7 *Syntax Directed Definition*

#	Production	Semantic Rule
1	a : b c d	d.i = f3(c.s,b.s)
2	b : B	b.s = f1(B.lexeme)
3	c : C	c.s = f2(C.lexeme)
4	d : D	

The VAL stack at the time of reduction of d → D is as shown in Fig. 4.7. The VAL stack contains the synthesised attributes c.s and b.s at that point of time. In general, the VAL stack contains the attributes of the left siblings of the parent during a reduction.

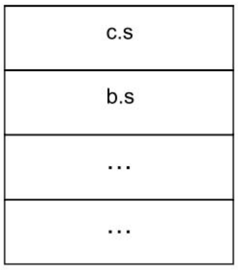


Fig. 4.7 *VAL stack contains attributes of left siblings of parent*

Let's say, we introduce a new ***marker symbol M*** generating ∊, in the existing grammar. The marker symbol M is inserted before 'd' in the RHS of production 1 to take advantage of the fact that both c.s and b.s are on the VAL stack. The semantic action for the production M → ∊ involves saving the values of the synthesised attributes c.s and b.s in variables 'save_c_s' and 'save_b_s' respectively by retrieving them from the VAL stack. The variables save_c_s and save_b_s can be later used for computing the attribute 'd.i' in production 4. Table 4.8 shows the transformed translation scheme including the production for M and its semantic action. This translation scheme can be used directly for evaluating the semantic rules during the bottom-up parsing.

Table 4.8 *Translation scheme using saved variables*

#	Production
1	a : b c **M** d { d.i = f3(saved_c_s,saved_b_s) }
2	b : B { b.s = f1(B.lexeme) }
3	c : C { c.s = f2(C.lexeme) }
4	d : D
5	M : Є { save_c_s = val[top] ; save_b_s = val[top-1] }

The introduction of a marker helped us evaluate inherited attributes depending on the synthesised attributes of its left siblings. The LR parser generators like 'yacc'/ 'bison' insert such markers like M implicitly and provide the user with a simplified notion of an embedded semantic action. Table 4.9 shows the translating scheme that can be used readily with LR parser generators like 'yacc'/ 'bison'. Observe the usage of $ variables to represent the elements on the VAL stack. The variable $2 represents the val[top] and $1 represents val[top-1] in this context.

Table 4.9 *Translation scheme compatible with 'yacc'/ 'bison'*

#	Production		
1	a	:	b c { saved_b_s=$1 ; saved_c_s=$2 } d { d.i = f3(saved_c_s,saved_b_s) }
2	b	:	B { b.s = f1(B.lexeme) }
3	c	:	C { c.s = f2(C.lexeme) }
4	d	:	D

Consider another translation scheme shown earlier in Table 4.6 (reproduced below for convenience) for reinforcing the ideas on the handling of inherited attributes during the bottom-up parsing. This translation scheme recognises variable declarations in C language and makes entries into symbol table. It is a manifestation of L-attributed syntax-directed definition shown in Table 4.3.

#	Production		
1	declaration_list	:	declaration_list declaration
2		|	declaration
3	declaration	:	type_spec { identifier_list.type = type_spec.data_type } identifier_list ';'
4	type_spec	:	INT { type_spec.data_type = INT }
5		|	CHAR { type_spec.data_type = CHAR }
6		|	FLOAT { type_spec.data_type = FLOAT }
7	identifier_list	:	{ identifier_list$_1$.type = identifier_list.data_type } identifier_list ',' IDENTIFIER { insert(IDENTIFIER.place,identifier_list.type) }
8	identifier_list	:	IDENTIFIER { insert (IDENTIFIER.place, identifier_list.type) }

The inherited attribute 'identifier_list.type' in the Production 3 cannot be filled in during bottom-up parsing, since the symbol 'identifier_list' is yet to be reduced. However, we can use a variable 'saved_identifier_list_type' to store the attribute value, so that it can be used later in Production 7 and 8. By using the variable 'saved_identifier_list_type' to store the attribute, the attribute identifier_list.type itself can be completely eliminated as shown below.

3	declaration	:	type_spec { ***saved_identifier_list_type*** = type_spec.data_type } identifier_list ';'
7	identifier_list	:	identifier_list ',' IDENTIFIER { insert(IDENTIFIER.place, ***saved_identifier_list_type***) }
8	identifier_list	:	IDENTIFIER { insert (IDENTIFIER.place, ***saved_identifier_list_type***) }

We can make use of the special variable $1 in the embedded semantic action of Production 3 to access val[top] containing 'type_spec.data_type'. The final translation scheme using the $ variable compatible with LR parser generators like 'yacc'/ 'bison' is shown in Table 4.10.

Table 4.10 *Translation scheme for C-declarations compatible with 'yacc'/ 'bison'*

#	Production		
1	declaration_list	:	declaration_list declaration
2		|	declaration
3	declaration	:	type_spec { ***saved_identifier_list_type*** = $1 } identifier_list ';'
4	type_spec	:	INT { type_spec.data_type = INT }
5		|	CHAR { type_spec.data_type = CHAR }
6		|	FLOAT { type_spec.data_type = FLOAT }
7	identifier_list	:	identifier_list ',' IDENTIFIER { insert(IDENTIFIER.place, ***saved_identifier_list_type***) }
8	identifier_list	:	IDENTIFIER { insert (IDENTIFIER.place, ***saved_identifier_list_type***) }

4.1.3.3 Example 1—Bottom-Up Translation This section demonstrates an example program that evaluates semantic actions during the bottom-up parsing using the theory described in the preceding section. The example implements the translation scheme presented in Table 4.10. The program shows the usage of the VAL stack and the special $ variables in LR parser generators like bison to help the evaluation of semantic rules. The program takes as input, a sample C program with some declarations of variables using the basic data types like 'int', 'char' and 'float'. The output of the example is symbol table entries generated from the processing of the declarations in the input C program. The dialog below shows the example program taking in C programs, and printing out the symbol table entry details.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -oc_decl_gram.cc c_decl_gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o c_decl_gram.o c_decl_gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc_decl_lex.cc c_decl_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o c_decl_lex.o c_decl_lex.cc

# Building ex1 Binary
$ g++ -g -Wall c_decl_gram.o c_decl_lex.o -o ex1

# This is a sample input source file
$ cat -n test1.c
    1 int a,b,c;
    2 float d,e,f;
    3 char i,j,k;

# Parsing and displaying Symbol table information for the declarations
$ ./ex1 test1.c
Identifier name=a type=INT
Identifier name=b type=INT
Identifier name=c type=INT
Identifier name=d type=FLOAT
Identifier name=e type=FLOAT
Identifier name=f type=FLOAT
Identifier name=i type=CHAR
Identifier name=j type=CHAR
Identifier name=k type=CHAR
SYNTAX CORRECT

# Input source file with Syntax Error
$ cat -n test1a.c
    1 int a,b c; /* Missing comma */

# syntax Error is detected
$ ./ex1 test1a.c
Identifier name=a type=INT
Identifier name=b type=INT
int a,b c
        ^

Error in Line number=1
SYNTAX INCORRECT
```

4.1.3.4 *Top-Down Translation* We saw in the previous section, how we can evaluate semantic rules during the bottom-up parsing of the input for L-attributed definitions. In this section, we discuss how we can evaluate semantic rules for L-attributed definitions during the top-down parsing of the input.

Consider the syntax-directed definition presented in Table 4.4 (reproduced below for convenience) for discussing about top-down translation. The SDD describes a desktop calculator that takes an expression involving constants as input and emits out the evaluated result. The input and the corresponding output for the SDD are given in Table 4.5.

#	Production			Semantic Rule
1	sentence	:	expr	print expr.value
2	expr	:	expr '+' term	expr.value = expr_1.value + term.value
3		|	expr '-' term	expr.value = expr_1.value - term.value
4		|	term	expr.value = term.value
5	term	:	term '*' factor	term.value = term_1.value * factor.value
6		|	term '/' factor	term.value = term_1.value / factor.value
7		|	factor	term.value = factor.value
8	factor	:	'(' expr ')'	factor.value = expr.value
9		|	CONSTANT	factor.value = CONSTANT.lexeme

We have studied earlier in the chapter on syntax analysis that left-recursive grammar cannot be parsed using top-down parsing method. The productions 2, 3, 5 and 6 above exhibit left recursion, we need to transform these productions to remove the left recursion. The transformation to eliminate the left recursion should also take into account the rearrangement of semantic actions.

Let's look at how to transform any left-recursive grammar and its semantic rules to be non-left recursive. Consider a left-recursive grammar shown in Table 4.11. In the syntax-directed definition, A.a, B.b , C.c are synthesised attributes, f1 and f2 are arbitrary functions.

Table 4.11 *A syntax-directed definition*

1	a	:	a B	a.a := f1(a1.a,B.b)
2	a	:	C	a.a := f2(C.c)

Table 4.12 shows the resultant translation scheme after the elimination of left recursion and the rearrangement of the semantic actions from the syntax-directed definition. This grammar recognises the same sentences as the original grammar.

Table 4.12 *Translation scheme*

1	a	:	C { a_rest.i = f2(C.c) } a_rest { a.a = a_rest.s }
2	a_rest	:	B { a_rest1.a = f1 (a_rest.i , B.b) } a_rest { a_rest.s = a_rest1.s }
3		:	Є { a_rest.s = a_rest.i }

Using the above principle, we eliminate left recursion in the Productions 2, 3, 5 and 6 of the syntax-directed definition of Table 4.4 and also transform it to a translation scheme. The resulting translation scheme suitable for top-down translation is shown in Table 4.13.

Table 4.13 *Translation scheme for the desktop calculator suitable for top-down parsing*

#	**Production**		
1	sentence	:	expr
2	expr	:	term { expr_rest.i = term.value } expr_rest { expr.value = expr_rest.s; }
3	expr_rest	:	'+' term { expr_rest_1.i = expr_rest. i + term.value } expr_rest { expr_rest.s = expr_rest_1.s }
4		|	'-' term { expr_rest_1.i = expr_rest. i - term.value } expr_rest { expr_rest.s = expr_rest_1.s }
5		|	ε { expr_rest.s = expr_rest.i; }
6	term	:	factor { term_rest.i = factor.value ; } term_rest { term.value = term_rest.s ; }
7	term_rest	:	'*' factor { term_rest_1.i = term_rest. i * factor.value } term_rest { term_rest.s = term_rest_1.s }
8		|	'/' factor { term_rest_1.i = term_rest. i / factor.value } term_rest { term_rest.s = term_rest_1.s }
9		|	ε { term_rest.s= term_rest. i }
10	factor	:	'(' expr ')' { factor.value = expr.value }
11		|	CONSTANT { factor.value = CONSTANT.lexeme }

Now that we have a translation scheme suitable for top-down translation in Table 4.13, the next step is to code a top-down translator that uses the translation scheme. We shall implement a recursive descent parser and include functionality of evaluating the attributes. We have seen in the previous chapter that a recursive descent parser is a collection of procedures one for each non-terminal. Each procedure is responsible for parsing the constructs defined by its non-terminal.

The following guidelines help us build a top-down translator from a translation scheme similar to the one we have in Table 4.13. These guidelines can be used for building a top-down translator for any translation scheme originating from a L-attributed definition.

For each non-terminal N, we construct a function N, that returns success or failure depending on whether it was able to expand the non-terminal correctly or not. The function N takes in formal parameters for each inherited attribute and the synthesised attributes of N. The inherited attributes are used in the function to compute the dependent attributes of N or any one of its children. The synthesised attributes that come in as formal parameters of N are usually passed by reference so that the value can be derived and populated inside the function. The function for N would typically have a local variable for each attribute of the grammar symbols that appear in a production for N. For example, consider Productions 2, 3 and 4 governing the non-terminal 'expr_rest'.

3	expr_rest	:	'+' term { expr_rest$_1$.i = expr_rest. i + term.value } expr_rest { expr_rest.s = expr_rest$_1$.s }
4		|	'-' term { expr_rest$_1$.i = expr_rest. i - term.value } expr_rest { expr_rest.s = expr_rest$_1$.s }
5		|	ε { expr_rest.s = expr_rest.i; }

Based on the guideline mentioned above, the Productions 3, 4 and 5 would result in the following function.

```
int expr_rest(int expr_rest_i,int &expr_rest_s)
{
   int value;
   int term_value;
   int expr_rest1_i;
   int expr_rest1_s;

   int op;

   if(match('+')){
      op='+';
   }else if(match('-')){
      op = '-';
   }else {
      expr_rest_s=expr_rest_i;
      return(SUCCESS);
   }

   if(term(term_value) == SUCCESS ){
      if(op == '+' ){
         expr_rest1_i = expr_rest_i + term_value ;
      }else{
         expr_rest1_i = expr_rest_i - term_value ;
      }

      if(expr_rest(expr_rest1_i,expr_rest1_s) == SUCCESS ){
         expr_rest_s = expr_rest1_s;
         return(SUCCESS);
      }
30     }
31
32     return(FAILURE);
33  }
```

Listing 4.1 *Code for the non-terminal 'expr_rest'*

The function expr_rest returns SUCCESS on being able to expand the non-terminal correctly and FAILURE on not being able to. The function expr_rest has two formal parameters, one being expr_rest_i (denoting expr_rest.i—the Inherited attribute of expr_rest) and the other expr_rest_s (denoting expr_rest.s – the synthesised attribute of expr_rest). The formal parameter 'expr_rest_s' is passed by reference, since its value is computed in the function and would be typically used by the caller of the function expr_rest. The inherited attribute expr_rest_i is used in computing expr_rest_s in line 15 (in accordance to Production 5) and computing expr_rest1.i in line 21 and 23 (in accordance to Productions 3 and 4). The synthesised attribute expr_rest_s is passed by reference and is populated in the lines 15 and 27 in cases where the function returns SUCCESS. The function expr_rest has 3 local variables term_value, expr_rest1_i and expr_rest1_s each denoting attributes for grammar symbols suggestive in the name. The code associated with each of the functions for the non-terminal is decided on the basis of the next token as explained in Section 3.5.1. The function expr_rest shown in Listing 4.1 reflects the principles of recursive descent parser discussed in Section 3.5.1.

For each terminal T, we derive the synthesised attribute 't' from the lexical analyser. We store the value of the synthesised attribute in the local variable declared. This is followed by a call to match the token and advance the input. Let's take Productions 10, 11 and their associated code shown below as a reference to understand this concept.

10	factor	:	'(' expr ')' { factor.value = expr.value }
11		:	CONSTANT { factor.value = CONSTANT.lexeme }

```
int factor(int &factor_value)
{

   int CONSTANT_lexeme;
   int expr_value;

   if(current_token == CONSTANT ){
      CONSTANT_lexeme = atoi(yytext);
      factor_value = CONSTANT_lexeme;
      match(CONSTANT);
      return(SUCCESS);
   } else if(match('(') ){
      if(expr(expr_value) == SUCCESS ){
         factor_value = expr_value;
         if(match(')') ){
            return(SUCCESS);
17            }
18         }
19     }
20
21     return(FAILURE);
22  }
```

Listing 4.2 *Code derived from production 10 and 11*

In line 8 of Listing 4.2, we derived the value of synthesised attribute—lexeme of the terminal CONSTANT from the lexical Analyser and stored it in the variable CONSTANT_lexeme declared for the attribute CONSTANT.lexeme. The Line 10 makes a call to function match, which matches the token and advances the input.

4.1.3.5 Example 2—Top-Down Translation This section demonstrates an example program that evaluates semantic actions during the top-down parsing using the theory described in the preceding section. The example implements the translation scheme presented in Table 4.13 to build a desktop calculator. The program shows the usage of the guidelines provided in the preceding section to construct a top-down translator for L-attributed definitions. The program takes as input an expression involving constants. The output of the example is the evaluated result of the input expression, similar to the desktop calculator. The dialog below shows the example program taking in expressions involving constants, and printing out the result of the expression.

```
# Generating the Lexical Analyzer from lexical Specifications
$ flex -otop_down_lex.cc top_down_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o top_down_lex.o top_down_lex.cc
top_down_lex.cc:1040: warning: 'void yyunput(int, char*)' defined but not used

# Building ex2 Binary
$ g++ -g -Wall ex2.cc top_down.cc top_down_lex.o -o ex2

# Executing it for a sample Expression
$ ./ex2 '9+15-20'
result=4
SYNTAX CORRECT

# Another sample Expression
$ ./ex2 '3*21 - (4*5)'
result=43
SYNTAX CORRECT

# Another sample Expression
$ ./ex2 '(9*53)/(7-4)'
result=159
SYNTAX CORRECT

# syntax Error in Expression
$ ./ex2 '9*53)/(7-4)'
SYNTAX INCORRECT
```

4.1.4 Parse Tree Method

In the parse tree method of evaluating semantic rules, we create a parse tree during the syntax analysis. Based on the parse tree and semantic rules, a dependency graph is created post-syntax analysis. Next, we evaluate the value of each of the attribute as signified by a node in the dependency graph. The order of evaluation of attributes is obtained from a topological sort of the dependency graph. The evaluation of attributes yields the necessary side-effect like addition into symbol table, generation of intermediate code, and so on.

Let's take the syntax-directed definition shown in Table 4.4 as an example for understanding the parse tree method of evaluating semantic rules. The SDD is for a desktop calculator taking expressions involving constants as input and printing out the result of the expression as output. Table 4.5 shows some sample inputs and the corresponding output given out by the syntax-directed definition.

The parse tree method of evaluating semantic rules consists of 3 steps as shown in Fig. 4.8.

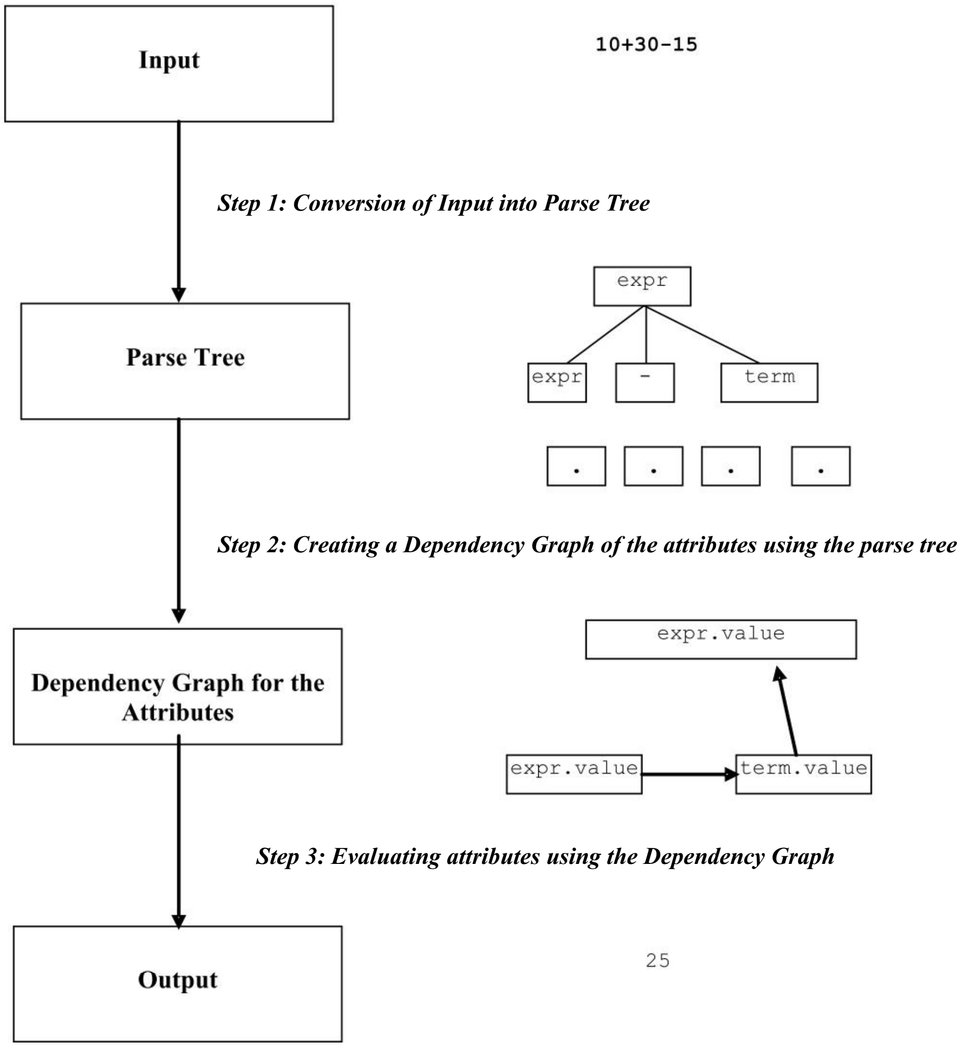

Input
10+30-15
Step 1: Conversion of Input into Parse Tree
Parse Tree
expr
expr
-
term
Step 2: Creating a Dependency Graph of the attributes using the parse tree
Dependency Graph for the Attributes
expr.value
expr.value
term.value
Step 3: Evaluating attributes using the Dependency Graph
Output
25

Fig. 4.8 *Steps in converting input-source program to intermediate code*

In the first step, the input is converted into a parse tree. In a parse tree, each of the interior nodes represents a non-terminal, while the leaf nodes denote terminals. The parse tree is an outcome of syntax analysis.

The parse tree contains nodes pertaining to each syntactic unit. For example, if an input source were parsed for a grammar shown in Table 4.4, the parse tree would contain nodes like 'factor', 'term', 'stmt', etc. The nodes in the parse tree are typically associated with attributes specific to the nature of the node. The 'expr' node is associated with an attribute 'value'. Now, the 'value' associated with the 'expr' node depends on its constituents like the 'value' of factor node and the 'value' of term node, and so on. The dependency of the attributes of various nodes dictates the order in which the attributes are computed. The second step involves creation of the dependency graph—a graph associating the order in which the attributes need to be computed for making the translation of input into output.

In the third step, we compute the values of attributes in the order specified by the dependency graph obtained earlier in step 2. The computation of values for the attributes is governed by semantic rules. The evaluation of semantic attributes generates the output.

Figure 4.8 shows parts of the sample output at each of the stages for the input '10+30–15'. The syntax-directed definition it conforms to is in Table 4.4.

The next three sections explain in detail, the three steps required for evaluating semantic rules using parse tree method as outlined in Fig. 4.8.

4.1.4.1 *Step 1—Conversion of Input Source Program into Parse Tree*

The first step involves converting the input into a parse tree. This is usually done during the parsing phase itself. Recall that the parser generators allow for program fragments to be executed when a production gets reduced (or expanded in case of a top-down parser). This feature can be extensively used for creating the parse tree.

Let's take the case of a bottom-up parser. Typically for a production P → A B C in the grammar, the program fragment to be executed while reducing a production for generating a parse tree would be as shown in Algorithm 4.2.

```
{
        Create a Node P
        Make the Nodes A,B,C as the children of P from left to right
}
```

Algorithm 4.2 *The program fragment to be executed for generating parse tree*

There are facilities in parser generators like bison/yacc for aiding the creation of parse tree. Typically a pointer to the newly created node P (in Algorithm 4.2) can be stored on the stack using the notation $$. Similarly, pointers to each one of the entities on the RHS can be retrieved from the stack by using the notation $1 for the first entity on the RHS, $2 for the second entity, and so on. These facilities in the parser generators are used in the parse tree creation.

In order to work on a parse tree (for e.g. printing the parse tree nodes on the screen or adding information to various nodes in the parse tree), we need to have an order of visiting (traversing) each of the parse nodes in the tree starting from the root. A common method of traversing the parse nodes is the depth-first traversal method discussed previously. It starts with the root and recursively visits the children of each node, left to right order. Figure 4.9 shows the order in which the parse tree nodes are visited by the depth-first traversal method with the node #1 being visited first followed by node #2 and so on in the series. This is a parse tree created for the input '10 + 30 – 15' conformant to the grammar in Table 4.4.

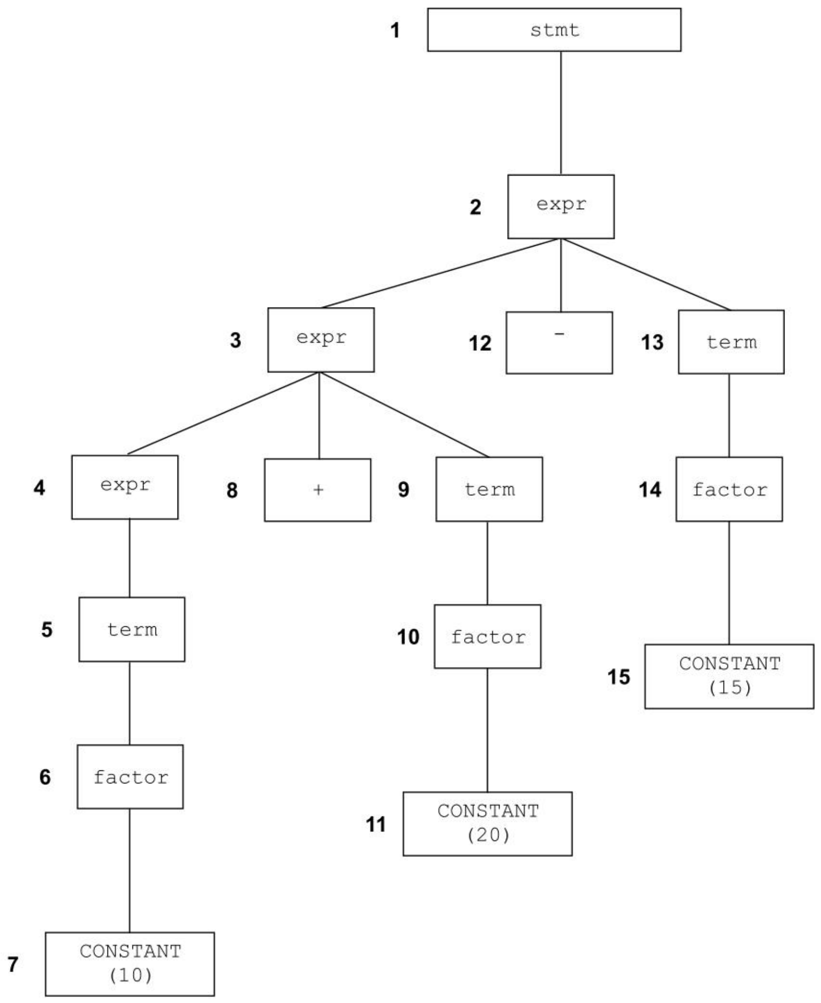


Fig. 4.9 *Parse tree*

The algorithm for depth-first traversal is formalised and shown in Algorithm 4.3. Observe the recursive nature of the algorithm.

```
procedure dfvisit ( node *n)
{
        process_the node (); /* For e.g. Printing the Node */

        for each of the child m of n from left to right
        {
             dfvisit(m);
        }
}
```

Algorithm 4.3 *Depth-first traversal*

In step 1 of the parse tree method, we learnt the concepts of creating the parse tree (Algorithm 4.2) and traversing the parse tree for, say, printing it (Algorithm 4.3). We now move on to the second step of the parse tree method of evaluating semantic rules—the creation of dependency graph.

4.1.4.2 *Step 2—Creation of Dependency Graph* We had seen earlier that the order in which the attributes are evaluated could be depicted by a directed graph called as dependency graph. In this section we learn how to create a dependency graph from the parse tree.

In a dependency graph, there is a node for each of the attribute associated with each of the nodes of the parse tree. For example, consider the parse tree in Fig. 4.9. Let us take the case of the expr node (Node 2). The expr node has one attribute called 'value'. The dependency graph will have one node, namely expr. value corresponding to the expr node numbered 2 in the parse tree. In a similar fashion, we would have a node expr.value in the dependency graph corresponding to the expr node numbered 3 in the parse tree. There would be another node term.value in dependency graph corresponding to the term node numbered 13 of the parse tree, and so on. Figure 4.10 shows the dependency graph nodes associated with the three parse tree nodes just discussed, namely—expr (Node 2) and its children (Node 3 and Node 13). The full dependency graph would contain nodes corresponding to all the parse nodes and all of its attributes.

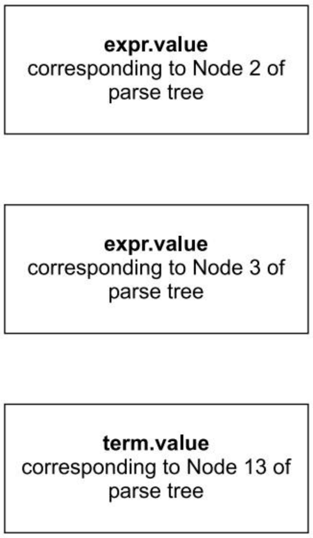

expr.value
corresponding to Node 2 of parse tree
expr.value
corresponding to Node 3 of parse tree
term.value
corresponding to Node 13 of parse tree

Fig. 4.10 *Some dependency graph nodes*

After creating a node in the dependency graph for each attribute corresponding to each parse tree node, the dependency nodes are 'chained' by drawing edges between them. The dependency graph has an edge from a dependency graph node 'a' to dependency graph node 'b', if 'b' depends on 'a'. For example, consider the dependency graph nodes shown in Fig. 4.10. The production used to reduce for creating the nodes 2, 3 and 13 is the production number 3. The Production 3 and the semantic rule from the SDD are reproduced here.

3	`expr : expr '-' term`	`expr.value = expr1.value - term.value`

Observing the semantic rule associated with the above production, we have:

- expr.value of Node 2 depends on expr.value of Node 3.
- expr.value of Node 2 depends on term.value of Node 13.

We create two edges in the dependency graph for the corresponding elements.

1. From expr.value of Node 3 to expr.value of Node 2.
2. From term.value of Node 13 to expr.value of Node 2.

The section of dependency graph corresponding to these three edges is shown in Fig. 4.11.

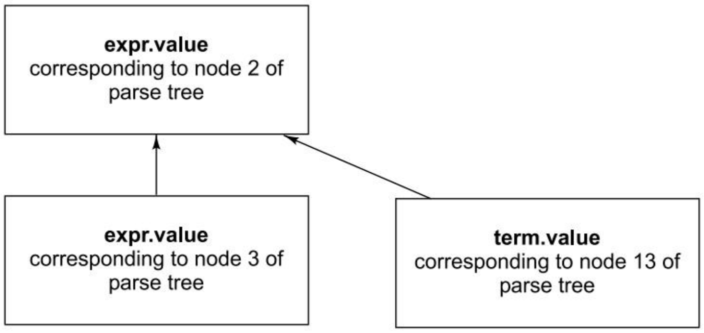

expr.value
corresponding to node 2 of parse tree
expr.value
corresponding to node 3 of parse tree
term.value
corresponding to node 13 of parse tree

Fig. 4.11 *A section of dependency graph*

In this manner the edges between the dependency graph nodes are created on the basis of dependencies between the attributes as specified in semantic rules.

To sum up this step, the creation of dependency graph is a two-part process:

1. Creation of dependency graph nodes for each one of the attributes corresponding to each of the parse tree nodes.
2. Creation of edges from one dependency graph node 'a' to another dependency graph node 'b', if 'b' depends on 'a'.

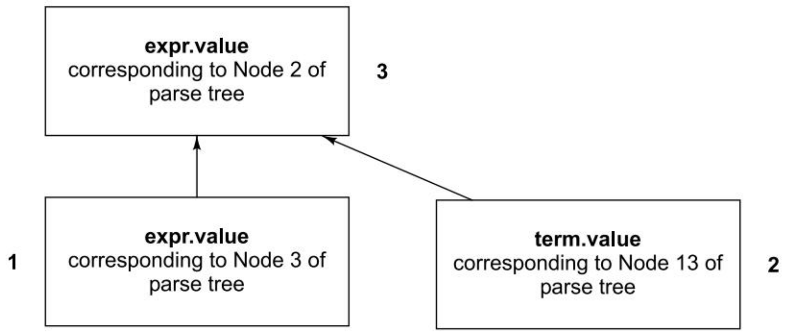
This procedure is formalised in Algorithm 4.4.

```
create_dep_graph()
{
        /* Creation of dependency graph nodes for each one of the attributes corresponding
        to each of the parse tree node */

        for each node n in a parse tree
             for each attribute a of the grammar symbol at n
                  Construct a Node in the dependency graph

        /* Creation of edges from one dependency graph node 'a' to another dependency
        graph node 'b', if b depends on a */

        for each node n in the parse tree
             for each semantic rule b := f (c1,c2,c3 …ck) associated with production used at n
                  for I:=1 to k
                       Construct an edge from the node for ci to the node b
}
```

Algorithm 4.4 *Creation of dependency graph*

The dependency graph created using Algorithm 4.4 gives the dependencies among the attributes of various parse tree nodes.

The order of evaluation of attributes is obtained by performing a topological sort of the dependency graph. A topological sort of a directed non-cyclic graph is any ordering *m*1, *m*2, *m*3,….*mk* of the nodes of the graph such that edges exist from earlier nodes to later nodes. It means that if an edge exists from *mi* to *mj*, then topological sort will have *mi* ahead of *mj*. As an example, the topological sort of the dependency graph discussed earlier is shown in Fig. 4.12. It would be interesting to note that the relative ordering between nodes marked 1 and 2 is not significant. The only criterion is that the nodes marked 1 and 2 have to be aligned earlier than the node marked 3. This is because there are edges 1 → 3 and 2 → 3.



expr.value corresponding to Node 2 of parse tree
3
1
expr.value corresponding to Node 3 of parse tree
term.value corresponding to Node 13 of parse tree
2

Fig. 4.12 *Topological sort of dependency graph*

Note that if there is a situation in which an attribute *x.a* depends on another attribute *y.b*, which in turn depends on another attribute say *z.c*. If *z.c* depends on the original attribute *x.a* then there is a cycle in the graph. The parse tree method of evaluating semantic rules does not work when there are cycles in the dependency graph.

An algorithm to perform topological sort of a directed non-cyclic graph is given in Algorithm 4.5.

```
TopSort(Graph G )
{
        for(counter=1;counter < number_of_nodes ; counter ++ ){
             V = find the graph node with incoming edges as 0
             if ( V is NULL ){
                  error – The Graph G has a Cycle - exit
             }
             Add V to the sorted list at the end
             for each graph node W adjacent to V {
                  W.incoming edges –
             }
        }
}
```

Algorithm 4.5 *Topological sort of non-cyclic directed graph*

The topological sort of the dependency graph yields the order in which the attributes associated with the nodes in a parse tree can be evaluated. For example, from the Fig. 4.12, the value attribute of the expr node corresponding to node 2 in the parse tree would be evaluated before the value attribute of expr node corresponding to node 3 in the parse tree.

4.1.4.3 *Step 3—Evaluation of Attributes using the Dependency Graph*

In step 3, we evaluate the value of each of the attributes as signified by a node in the dependency graph. The order of evaluation of attributes was obtained from a topological sort of the dependency graph in the previous step. The evaluation of attributes yields the output.

The value for each of the attributes represented by a node in the dependency graph is computed using the related semantic rule. As an example, consider the section of dependency graph shown in Fig. 4.12. The dependency graph node marked 3, which represents *expr.value*, has originated from the parse tree node 2. The node 2 in the parse tree has been created due to reduction by Production 3 (expr → expr '–' term). The semantic rule for the Production 3 is *expr.value* = *expr1.value* – *term.value.* We know that *expr1.value* (dependency graph node 1) and term.value (dependency graph node 2) would have already been evaluated before we attempt to evaluate *expr.value* (dependency graph node 3) because the evaluation order is based on the topological sort of the dependency graph.

The evaluation of all the attributes represented by the entire set of nodes in dependency graph populates all the attributes of parse tree nodes. The evaluation of attributes also includes invocation of routines causing important side-effects like, say, adding an entry into symbol table, printing out a value, etc.

This step concludes the parse tree method of evaluating semantic rules.

4.1.4.4 *Example 3—Parse Tree Method*

This section demonstrates an example program that evaluates semantic actions using the parse tree method for the syntax-directed definition shown in Table 4.4. In this example we create the parse tree, compute the dependency graph, topologically sort it and evaluate the attributes for generating the output. The program takes as input an expression involving constants. The output of the example is the evaluated result of the input expression, similar to the desktop calculator. The dialog below shows the example program taking in expressions involving constants, and printing out the result of the expression.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -optree_gram.cc ptree_gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o ptree_gram.o ptree_gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -optree_lex.cc ptree_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o ptree_lex.o ptree_lex.cc

# Building ex3 Binary
$ g++ -g -Wall ex3.cc ptree_gram.o ptree_lex.o ptree_method.cc -o ex3

# Executing it for a sample Expression
$ ./ex3 '9+15-20'
SYNTAX CORRECT
4

# Another sample Expression
$ ./ex3 '3*21 - (4*5)'
SYNTAX CORRECT
43

# Another sample Expression
$ ./ex3 '(9*53)/(7-4)'
SYNTAX CORRECT
159

# Syntax Error in Expression
$ ./ex3 '9*53)/(7-4)'
9*53)/(7-4)
    ^

syntax error
SYNTAX INCORRECT
```

4.1.5 Rule-based Method

In rule-based method, the evaluation order of the attributes at each parse tree node is determined by means of analysing the semantic rules at the time of constructing the compiler. The order of visiting the children nodes for evaluation of semantic rules for a given parse tree node and the associated production is hard-coded into the compiler for evaluation of the semantic rules. Since the order of the evaluation is known at the time of compilation itself, there is ***no need for creating a dependency graph***. In order to completely evaluate all the semantic rules, it is possible that the parse tree is walked through multiple times in the rule-based method. In this section, we study about a rule-based method for evaluating semantic rules popularly known as ***recursive evaluator***.

In a recursive evaluator, we create an explicit parse tree and traverse it for evaluating the semantic rules by mutually recursive functions. The parse tree creation happens during the parsing and the traversal of the parse tree nodes for evaluating semantic rules happens later. The order in which we traverse the tree is determined by the analysis of the production and the semantic rules of that particular node *before the compilation itself*. It is possible that for evaluating semantic rules at one of the parse tree nodes, the children might be visited from the right-to-left. For evaluating semantic rules at another parse tree node, the children might be visited from the left-to-right. The main considerations involved in arriving at the correct order for the traversal of children for a parse tree node are (a) all the inherited attributes at a node are computed before the first visited (b) the synthesised attributes must be computed before we leave the node for the last time.

Let's take the SDD for the declaration statements in Pascal language in Table 4.14 as an example for understanding the intricacies of the recursive evaluator. The SDD is reproduced here for convenience.

Table 4.14 *Syntax-directed definition for processing of Pascal declarations*

#	Production			Semantic Rule
1	declaration_list	:	declaration_list declaration	
2		|	declaration	
3	declaration	:	identifier_list ':' type_spec ';'	identifier_list .type = type_spec. data_type
4	type_spec	:	INTEGER	type_spec. data_type = INTEGER
5		|	CHAR	type_spec. data_type = CHAR
6		|	REAL	type_spec. data_type = REAL
7	identifier_list	:	identifier_list ',' IDENTIFIER	identifier_list$_1$.type = identifier_list.type add_to_sym_table (IDENTIFIER.place, identifier_list.type)
8		|	IDENTIFIER	add_to_sym_table (IDENTIFIER.place, identifier_list.type)

We have selected the above grammar, which is not L-attributed, so as to bring out the fact that the recursive evaluator can also work for non-L-attributed grammars.

The first step in a recursive evaluator is the creation of a parse tree. The concepts and the method to obtain the parse tree have already been explained in Section 4.1.4.1. We skip the step in the discussion here.

The next step in the recursive evaluator is to traverse the parse tree to evaluate semantic rules. This is done by associating each of the non-terminals with a ***translation function***. This function performs the translation by visiting the children of the node in some particular order as determined by the production at the node before the compilation. For example, consider the translation function at the 'declaration' node. The translation function at declaration_node would visit its right child—the type_spec node and invoke its translation function to get the value of the data_type attribute. The category attribute is then passed down to its left child, the identifier_list node and its subtree by inheriting it as shown in Listing 4.3.

```
 1 void declaration_node::eval_semantic_rules()
 2 {
 3      int type_spec_data_type;
 4      identifier_list_node *identifier_list_ptr;
 5      type_spec_node *type_spec_node_ptr;
 6
 7      if(rule_no == 3 ){
 8
 9          /* declaration -> identifier_list ':' type_spec ';' */
10
11          identifier_list_ptr = (identifier_list_node *)children[0];
12          type_spec_node_ptr = (type_spec_node *)children[1];
13
14          /* Getting the 'data_type' synthesised from type_spec_node
15             - passing by reference
16          */
17          type_spec_node_ptr->eval_semantic_rules(type_spec_data_type);
18
19          /* Passing the 'data_type' to the children of identifier_list */
20          identifier_list_ptr -> eval_semantic_rules(type_spec_data_type);
21      }
22 }
```

Listing 4.3 *Evaluation of semantic rules for 'decl' node*

There are several interesting observations that can be made from Listing 4.3, highlighting the features of the recursive evaluator.

- At the highest level, we typically invoke the method to evaluate the semantic rules of the root of the tree. It internally evaluates the semantic rules of all its children. The children would similarly invoke the semantic rules for its children, and so on. In short, the evaluation of attributes happens in a recursive manner giving this the name—recursive evaluator.
- A non-terminal node can be created by one or more productions. The evaluation of semantic rules for a particular non-terminal node is determined by the production that created the node. Line 7 in Listing 4.3 illustrates this.
- The evaluation order does not depend on the order in which the parse tree nodes are created. This gives flexibility to visit the children in any order that is useful for evaluating the attributes. In the declaration_node evaluation shown in Listing 4.3, we are visiting the right child (type_spec) first and then the left child (identifier_list) . The reason in choosing this particular order is that we need to first fetch the value of data_type (line 17) to be passed on to identifier_list and its children (line 20).
- The evaluation function for each non-terminal N exhibits similarity with the top-down translator functions that we talked of in Section 4.1.3.4. Both of these functions take in formal parameters for each inherited attribute and the synthesised attributes of N. The inherited attributes are used in the function to compute the dependent attributes of N or any one of its children. The synthesised attributes that come in as formal parameters of N are usually passed by reference so that the value can be derived and populated inside the function. In Listing 4.3, the declaration_node does not have any inherited attributes and hence does not have formal parameters. However, identifier_list has inherited attributes and has a formal parameter. The type_spec node returns a synthesised attribute (data_type) by using call-by-reference technique.
- The flexibility of visiting the children in any order allows us to perform translation for any kind of grammar including the non-L attributed ones like the one above.

In the next section, we see the implementation of recursive evaluator for processing the SDD for Pascal declarations (Table 4.14) based on the concepts learnt here.

4.1.5.1 *Example 4—Recursively evaluating translator* This section demonstrates an example program that evaluates semantic actions using a recursively evaluating translator. The example implements the non-L-attributed syntax-directed definition shown in Table 4.14. The program takes as input, a sample Pascal program with some declarations of variables using the basic data types like 'integer', 'char' and 'real'. The output of the example is the symbol table entries generated from the processing of the declarations in the input Pascal program. The dialog below shows the example program taking in Pascal programs, and printing out the symbol table details.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -opas_decl_gram.cc pas_decl_gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o pas_decl_gram.o pas_decl_gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -opas_decl_lex.cc pas_decl_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o pas_decl_lex.o pas_decl_lex.cc

# Building ex4 Binary
$ g++ -g -Wall ex4.cc pas_decl_gram.o pas_decl_lex.o recur_eval.cc -o ex4

# This is an input source file
$ cat test4.pas
a,b,c,d,e:integer;
f,g,h,i,j : real ;
k,l,m,n,o : char ;

# Symbol Table for it
$ ./ex4 test4.pas
SYNTAX CORRECT
Identifier name=a type=INTEGER
Identifier name=b type=INTEGER
Identifier name=c type=INTEGER
Identifier name=d type=INTEGER
Identifier name=e type=INTEGER
Identifier name=f type=REAL
Identifier name=g type=REAL
Identifier name=h type=REAL
Identifier name=i type=REAL
Identifier name=j type=REAL
Identifier name=k type=CHAR
Identifier name=l type=CHAR
Identifier name=m type=CHAR
Identifier name=n type=CHAR
Identifier name=o type=CHAR
```

4.1.6 Comparison of Translation Methods

We had discussed about the three methods of evaluating semantic rules in the previous sections. Let's quickly reflect on each of these methods as to how they fare vis-à-vis various parameters. The following matrix provides the details of the comparison, where the methods are lined up as columns, and the parameters of comparison are marked up as rows.

Parameter	Translation interleaved with parsing	Parse tree method	Rule-based method
Principle on which the method is based on	Does not create a parse tree, does not create dependency graph. Evaluation of semantic rules happens during parsing.	Creates a parse tree, makes a dependency graph, evaluates the attributes based on topographic sort of the dependency graph.	Creates a parse tree, traverses it in proper order to evaluate the attributes and realise the SDD. The proper order is determined before compilation, by analysing the attributes and productions.
Attributes evaluation order	During symbol expansion for top-down translation. During symbol reduction for bottom-up translation.	Determined during the compilation by means of dependency graph.	The evaluation order is determined before compilation, by analysing the productions and hard-coding it in the recursive translator.
Applicability	L-attributed definitions.	Can be applied to any grammar provided there are no cycles in the dependency graph.	Any grammar.
Efficiency	Efficient, since we do not have to create a parse tree or dependency graph at the compile time.	Creation of dependency graph and sorting it reduces the efficiency.	Efficient, since we do not create a dependency graph at the compile time. However, multiple passes could reduce the efficiency.

4.2 SEMANTIC ANALYSIS

Armed with the concepts of syntax-directed translation, we now come back to the primary theme of the chapter—Semantic Analysis. As mentioned earlier, semantic analysis revolves around 2 tasks.

1. Process the declarations (e.g. variable declarations, function prototype declarations, type definitions, etc.) and build/update the symbol table to record the type information associated with various identifiers.
2. Examine the rest of the program to ensure that the identifiers are used correctly adhering to the type compatibility conventions defined by the language. For example, in an assignment statement, the types of the LHS and the RHS should match, the variable names used should be unique in that particular scope, and so on. This verification is called as ***type checking***.

Section 4.2.1 describes the first task in detail. In that, we present the productions and semantic actions of a semantic analyser that are relevant for the creation of a symbol table from a given input C program. Section 4.2.2 describes the second task, i.e. type checking in detail. In that, we examine the productions and semantic actions of a semantic analyser that are relevant for the type-checking and reporting of semantic errors in input C programs.

4.2.1 Processing Declarations

Let's start off the discussion on the declarations by looking at a simple declaration in C language of the form.

Table 4.15 *Declaration*

```
int counter , total ;
```

This declaration establishes the intent of the programmer to use the variables 'counter' and 'total' as integers. Most of the compiled languages require that the programmer specify the name and type of all the variables in the program before use. The main idea behind the declarations is to establish the 'data type' for every identifier used in the program, so that it can be verified for type correctness in the rest of the program. The function prototypes like the one seen at Table 4.16 extend the same idea to specify the data type of each of the parameters and the return value of the function.

Table 4.16 *Function prototype*

```
int my_func(int,float,char);
```

Establishing the data type of all the identifiers and updating the symbol table about the same is the primary purpose of processing the declarations. We will study about the data types and its representation in Section 4.2.1.1. The symbol table in which we record information about all the identifiers and their respective data types is studied in detail in Section 4.2.1.2. A translation scheme that can process an input file containing C language declarations and create a symbol table is discussed in Section 4.2.1.3. We follow it up with an implementation of the translation scheme and test it on sample C programs containing different types of declarations in Section 4.2.1.4.

4.2.1.1 Data Type A data type in a programming language can be a basic data type or a constructed one. In the declaration of Table 4.15 we are using one of the basic data types in C language—int. The other basic data types in C language are 'char', 'float', etc. A constructed data type is constructed from aggregates of the basic data type. Structures, arrays, functions are some of the examples of constructed data types in C language. The following declaration shows a variable *v*1 belonging to a constructed data type, namely a structure.

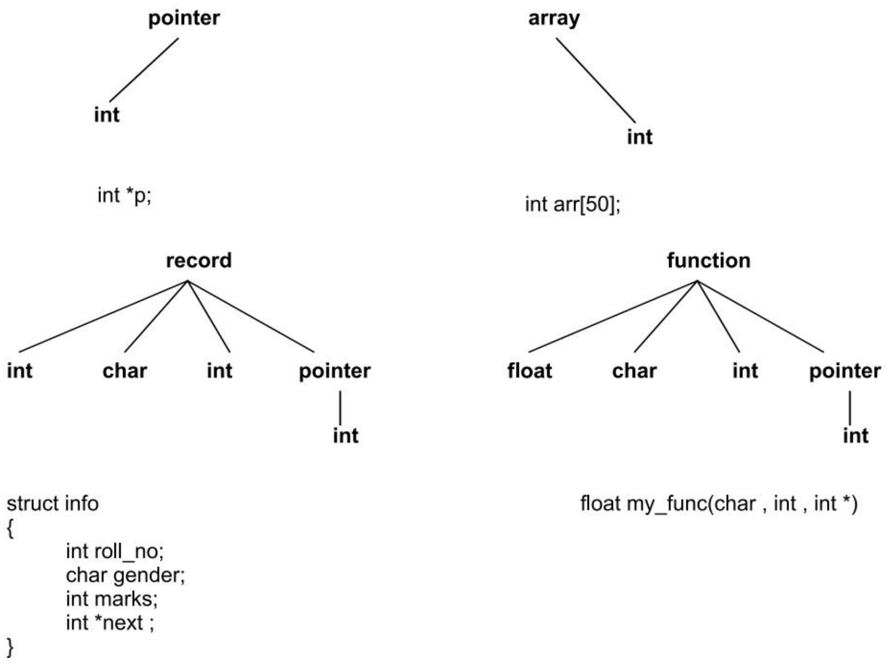
Table 4.17 *Declaration using a constructed data type—structure*

```
struct my_tag
{
       int a ;
       char b;
       float c ;
} v1;
```

A data type in a programming language is associated with a set of values and a set of operators/ operations allowed on those values. In the C language declaration shown in Table 4.15, 'counter' is an Integer data type. It can take any Integer value within a range. The set of operations allowed on it are +, –, *, etc. In the case of declaration in Table 4.17, we can use the dot (.) operator on the variable *v*1 to access the fields *a*, *b* or *c* as *v*1.*a* or *v*1.*b* or *v*1.*c* in any of the execution statements.

The data type information in a compiler cannot be represented by a simple integer, because we will have to denote constructed types like pointers, records, arrays, arrays of records and many other permutations and combinations of these. In order to represent all the basic types, constructed types and various combinations of basic and constructed types and of data, the type information needs to be more elaborate.

A data type in a compiler is represented by a ***type expression***. A basic data type of the language like int or char or float in C would be represented by a simple type expression. Constructed data types are represented by type expressions in conjunction with ***type constructors*** such as array, pointer, record and functions. A tree consisting of basic type expressions as leaves and type constructors as interior nodes can be used for visually representing any of the constructed data types. Figure 4.13 shows a few declarations of identifiers and their corresponding ***type expression tree***.



```
int *p;
```

```
int arr[50];
```

```
struct info
{
        int roll_no;
        char gender;
        int marks;
        int *next ;
}
```

```
float my_func(char , int , int *)
```

Fig. 4.13 *Type expressions for declarations*

A possible implementation of the data types is having a linked list of type expressions and type constructors as shown in Fig. 4.14. Each node in the linked list is a type constructor or a basic type expression. The node corresponding to the type constructor for the respective type would consist of the relevant attributes. For example, the node corresponding to the type constructor array would have an attribute 'no_of_elements', which would be indicative of the number of the elements in the array. The field 'next' in each of the node in linked list points to the next node in the type expression chain.

The linked list approach for implementing data type has been chosen in the example semantic analyser that follows in a section.

4.2.1.2 ***Symbol Table*** The main aspect of processing declarations in a compiler is to store the variable name along with its type information, scope information and other attributes in a ***symbol table***.

A symbol table is one of major data structures in a compiler, which is consulted and updated in most of the phases. Each entry in a symbol table corresponds to an identifier. The information in a symbol table entry is filled in different phases. For example, the lexical analysis phase might create the symbol table entry, while the semantic analysis phase might add the type information to the symbol table entry. The processing of declarations in semantic analysis phase usually results in adding the type information to a symbol table entry. Additional information is entered in the symbol table entry, whenever the role of the Identifier becomes clear.

A symbol table entry has fields for the name of the variable and the type of the variable (a pointer to its type expression tree). The symbol table entry also has a field 'offset' that specifies the relative address of the variable in a memory layout. For example, if there were 3 variables *v*1, *v*2 and *v*3 of size 4 bytes each declared in the same scope, the offset of *v*1 would be 0 and *v*2 would be 4 and *v*3 would be 8 bytes. There are other segments of information in the symbol table entry that would be very useful in code generation like whether a variable is a formal parameter to the function, the memory corresponding to the variable name, and so on.

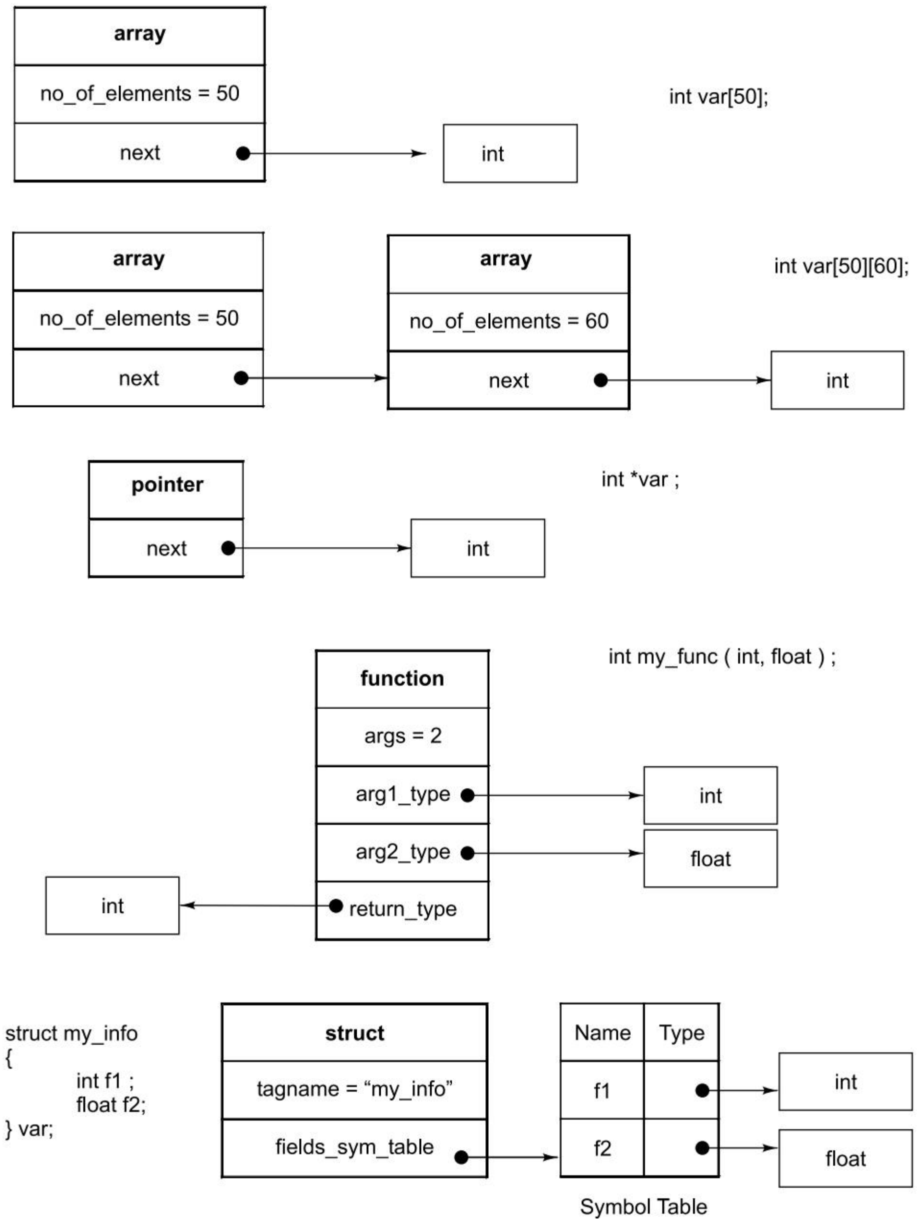


Fig. 4.14 *Linked list implementation of data type*

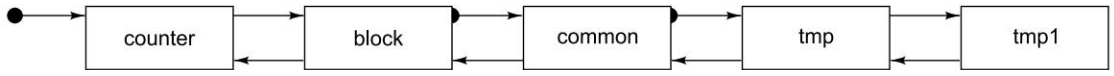
There are 3 main interface functions to the symbol table. They are:

1	insert	:	The insert operation inserts a symbol into the symbol table.
2	lookup	:	The lookup operation is used for looking up a given symbol in the symbol table
3	delete	:	The delete operation removes the entry from the symbol table.

Due to the high frequency of access, the symbol table needs to be maintained in a data structure that allows us quick insertion and fast lookup.

A doubly linked list of the records is a simple method to implement a symbol table. Each record is a symbol table entry containing among other things, an array for storing the name. New names are added to the end of the list by the insert operation. The 'lookup' operation proceeds backwards from the end of the list to the start, searching for the required identifier.

```
int counter, block, common, tmp, tmp1;
```



A ***hash table*** is another data structure commonly used to implement symbol table. The hash table allows us to insert, lookup and delete a symbol entry in almost constant time. A hash table is an array of entries called ***buckets***. The array can be indexed by an integer ranging from 0 to table size – 1. A ***hash function*** converts the search key into an integer in the range of 0 to (table size – 1). For a symbol table, the search key is the name of the identifier. When the hash function converts two or more of the search keys (identifier names in case a symbol table) into the same index of the hash table, a hash collision happens. A common collision resolving mechanism is to make a linked list of identifiers for each bucket instead of a single identifier.

Let's take a simple example to understand the terms used in the hash implementation for a symbol table. Figure 4.15 illustrates a hash table implementation, and shows a C language declaration of 5 identifiers and the corresponding hash table for the symbol table. The hash function used here is the ASCII value of the first letter in the identifier. The ASCII value of 'b' is 98, 'c' 99 and 't' is 116. The collision is resolved using a linked list of identifiers for each bucket. The buckets contain a pointer to the linked list of Identifiers.

The hash table size is 256. The lookup function would fetch the ASCII value of the first letter in the identifier and then index into the hash table to get the start of the linked list. The linked list is searched for obtaining the required identifier.

The efficiency of a symbol table implementation is usually judged by the time taken to insert an entry, lookup an entry and delete an entry in a symbol table. In a linked list implementation, the insert operation is constant in time, because we always insert at the end of the linked list. The lookup and delete operations are linear in time with the size of the linked list. The hash table implementation provides an almost constant in time performance for all the 3 operations of insert, lookup and delete of a symbol table entry. The performance of these operations does depend on the hash function to a large extent. A hash function that shows lesser rate of collisions would be more effective than the one that has higher rate of collisions.

Most of the compiled languages support the use of the same variable name in different scopes. The symbol table implementation should allow for the fetch of the symbol table entry in tune with the scope rules of the language. For example, C language supports the most closely nested scope rule. The listing 4.4 shows a C program in which a variable 'counter' is used in the function 'my_func'. The same variable has been declared and used in multiple scope blocks. In the C compiler, when the symbol table entry is looked up for the variable 'counter', it should correspond to the most closely nested scope. In the listing

4.4, when we are trying to resolve the symbol table entry during the processing of line number 9, it should correspond to the symbol table entry made on the processing of the declaration at line number 7 (as opposed to line number 3). Similarly, when the line number 20 is processed, the symbol table entry for 'counter' corresponding to the declaration on line number 18 should be fetched.

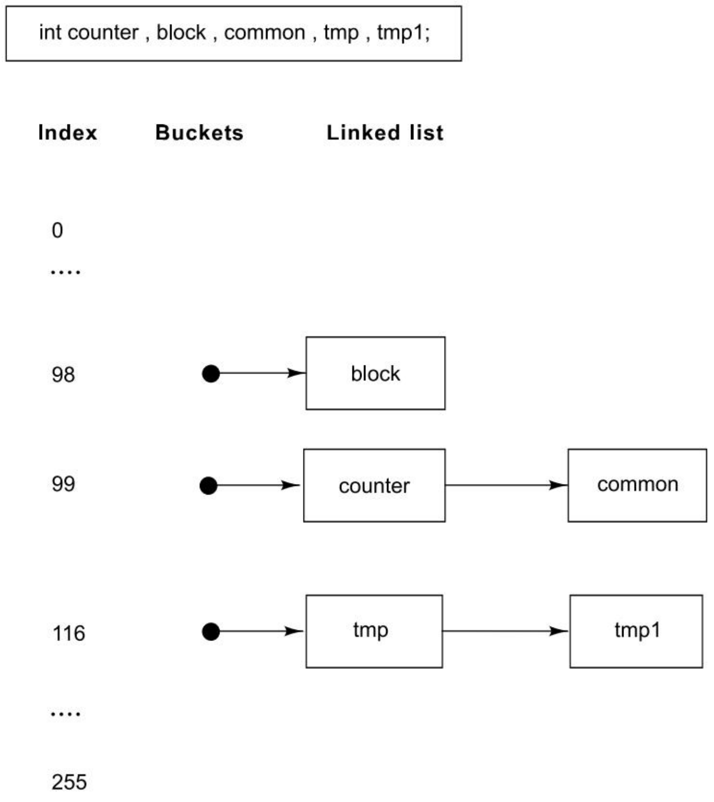


Fig. 4.15 *Hash table implementation*

```
1   #include <stdio.h>
2
3   int counter;
4
5   int my_func()
6   {
7       int counter, j;
8
9       j=counter;
10      for(counter=0; counter < 100 ; counter++)
11      {
12          int k, counter;
13
14          k=counter;
15
16          for(counter = 0; counter < 30 ; counter++ )
17          {
```

```
            int m, counter;

            m=counter;
            for(counter=0; counter < 20; counter++)
            {
                printf("Hi");
            }

        }
    }
    return(0);
}
```

Listing 4.4 *A C program*

A simple strategy to support the most closely nested scope in a compiler is to maintain a separate symbol table for each scope. The symbol table for the local block or a procedure is consulted first before checking the enclosing scope. Another strategy to support the most closely nested scope in a compiler is to number the blocks and procedures. The symbol table entry would be obtained by using the tuple consisting of the name of the variable, procedure number, and block number.

4.2.1.3 Translation Scheme In this section, we look at some of the productions and the semantic actions pertaining to the processing of the declarations in a semantic analyser for C language. These productions and semantic actions reinforce the ideas on symbol tables, type expressions and other relevant concepts that we studied previously. The translation scheme is presented in a pseudo-code form. The complete implementation of a semantic analyser for C language is presented in Section 4.3.

We start off by looking at some of the key productions and their semantic actions associated with the processing of declarations for a C language program as input.

The 'declaration' non-terminal is used for recognising a single declaration. It consists of a 'type_spec' and 'declarator_list' as given by Production 8. Some of the input code strings that match the declaration non-terminal are given in Table 4.18

8	declaration	:	type_spec declarator_list ';'

The 'type_spec' denotes the type specification in a declaration. Some of the input code strings that match the 'type_spec' non-terminal are given in Table 4.18. The Productions 10, 11 and 12 represent cases when the type specification is a basic data type. The lexical analyser provides the tokens INT, CHAR and FLOAT corresponding to the keywords 'int', 'char' and 'float'.

10	type_spec	:	INT {
			 `type_spec.data_type = new integer() ; /* Type expression */`
			 }

11		|	CHAR {
			 `type_spec.data_type = new character() ; /* Type expression */`
			 }

12		|	FLOAT {
			 `type_spec.data_type = new real(); /* Type expression */`
			 }

Production 14 represents a case when the type specification is a constructed data type—struct. Productions 15 and 16 are a couple of supportive productions for the same. Some of the input code strings that match the 'type_spec' non-terminal are given in Table 4.18. The marker 'T' is used as a placeholder to create a new symbol table and also record the tag name of the structure in a global hash table called 'type_list'.

14	type_spec	:	struct_specifier { `/* Type expression */` `type_spec.data_type = struct_specifier.data_type ;` }
15	struct_specifier	:	STRUCT IDENTIFIER '{' T declaration_list '}' { `t = type_list[IDENTIFIER.name]` `struct_specifier.data_type= t ;` `curr_sym_tab_ptr = tbl_stk.top();` `tbl_stk.pop();` }
16		|	STRUCT IDENTIFIER { `t = type_list[IDENTIFIER.name] ; /* Type */` `struct_specifier.data_type = t ;` }
29	T	:	∈ { `id = val[top-1]` `tbl_stk.push(curr_sym_tab_ptr);` `curr_sym_tab_ptr = new sym_tab() ;` `curr_sym_tab_ptr->previous = tbl_stk.top() ;` `t1 = type_list[id->name.c_str()];` `if(t1 == NULL){` `t1 = new record(curr_sym_tab_ptr,id->name);` `type_list[id->name]= t1;` `}else{` `/* Already present */` `print("struct %s defined multiple times \n",id->name);` `exit(0);` `}` }

Productions 17, 18 and 19 define the 'declarator'. Production 17 is used for supporting an array declaration. Observe that the left-recursive 'declarator' definition allows it to declare a multi-dimensional array or say a pointer with double level of indirection. Some of the input code strings that match the 'declarator' non-terminal are given in Table 4.18.

A 'declarator_list' is used to represent a list of variables following a type specification. Some of the input code strings that match the 'declarator_list' non-terminal are given in Table 4.18. The semantic action for both of them involves adding the variable into the current symbol table. The data type is fetched from the VAL stack at a known place. The semantic actions on Productions 25 and 26 also show the detection of a semantic error, namely the re-declaration of a variable in the same scope.

17	declarator	:	IDENTIFIER { `declarator.name = IDENTIFIER.name ;` }

```
18            |   declarator '[' CONSTANT ']' {
                       t = new array(CONSTANT.val) ; /* Type Constructor*/
                       if(declarator1.data_type != NULL )
                            declarator1.data_type->chain(t)
                       }else{
                            declarator1.data_type=t;
                       }
                       declarator = declarator1 ;
                  }
19            |   '*' declarator {
                       t = new pointer() ; /* Type Constructor */
                       if(declarator1.data_type != NULL )
                            declarator1.data_type->chain(t)
                       }else{
                            declarator1.data_type=t;
                       }
                       declarator.type = declarator1.type ;
                  }
```

```
25  declarator_list  :  declarator {
                            /* Inheriting attributes of type_spec
                                (see Production 8 ) on stack */
                            t = val[top-1].data_type ;

                            if(declarator.data_type != NULL )
                                declarator.data_type->chain(t)
                            }else{
                                declarator.data_type=t;
                            }

                            if(curr_sym_tab_ptr->find(declarator1.name) == NULL ){
                                curr_sym_tab_ptr->add(declarator.name,declarator. data_type);
                            }else{
                                print ("Re-declaration of the Variable '%s' in the same
                            scope line=%d \n",dptr->name,line_no);
                            }

                        }
26                   |  declarator_list ',' declarator {
                            /* Inheriting attributes of type_spec
                                (see Production 8 ) on stack */
                            t = val[top-1].data_type ;

                            if(declarator1.data_type != NULL )
                                declarator.data_type->chain(t)
                            }else{
                                declarator.data_type=t;
                            }

                            if(curr_sym_tab_ptr->find(declarator1.name) == NULL ){
                                curr_sym_tab_ptr->add(declarator.name,declarator.data_type);
                            }else{
                                print ("Re-declaration of the Variable '%s' in the
                        same scope line=%d \n",dptr->name,line_no);
                            }
                        }
```

The Table 4.18 shows some of the non-terminals and the input code strings that are generated by the non-terminal. The corresponding production used for generating the code string is also provided for getting a clear understanding.

Table 4.18 *Some non-terminals and the matching input code strings*

#	Non-terminal	Sample code strings matching the non-terminal	Production	Comments
1	declaration	int a1,a2,a3 ;	8	
2	declaration	char c1,c2;	8	
3	declaration	struct abcd { int a;int b; } x,y,z;	8	
4	declaration	struct efgh tmp;	8	
5	type_spec	int	10	
6	type_spec	char	11	
7	type_spec	float	12	
8	type_spec	struct abcd { int a;int b; }	14	'abcd' is called as tag name
9	type_spec	struct efgh	14	'efgh' is called as tag name
10	declarator	v1	17	
11	declarator	v1[100]	18	
12	declarator	v1[100][200]	18	
13	declarator	*v2	19	
14	declarator_list	v1,v2,v3	26	
15	declarator_list	v4, v5[10][20], *v6	26	

The type expressions are implemented using a linked list as shown in Fig. 4.14. A base class 'type_expr' having a member 'next' for pointing to the next element in the type expression chain is used. A method 'chain' of the type_expr class is used to insert an element at the end of the linked list. There are derived classes from type_expr for the basic data types like integer, character and real. The constructed data types like record (for structure), array, function are also derived from the base class 'type_expr'. The array class contains member to hold the number of elements as signified in Fig. 4.14. The record type expression contains an attribute 'fields'—a pointer to the symbol table of the fields associated with the structure. The other attribute for the struct type expression is 'tagname', the tag name used for defining more variables of the same structure type. A global data structure type_list contains the association between the structure's tag and the data type.

The scope of the declarations is supported by means of having multiple symbol tables, one for each function block encountered and a global symbol table for the global variables. The translation scheme uses tbl_stk—stack of pointers to symbol tables to manage the adding of a variable in appropriate symbol table. At the beginning of a structure definition (Production 15 and 29) we use a marker T to push the current symbol table pointer on the tbl_stk and start with a new symbol table. The 'previous' field of the current symbol table is used to store a pointer to the older symbol table. The same concept of tbl_stk can be used for handling any declaration nested by a pair of braces, but it is left as an exercise for the reader.

The symbol table maintenance is handled by the sym_tab class. It has methods add—for adding to the symbol table, find—for finding a symbol in the table, and print—for printing the symbol table. The variables (symbols) are added into the symbol table while handling every declarator in a declaration

(Production 25 and 26). The fields in a structure definition in the input source are added into a symbol table (attribute ‘fields’ of struct type expression) in Production 15.

4.2.1.4 *Example 5—Declarations* This section demonstrates an example program that implements the translation scheme, part of which was described in Section 4.2.1.3 to build the symbol table from the declarations. The program takes as input, a sample C program with some declarations of variables using the basic data types (e.g. int) and some constructed data types (e.g. struct). The output of the example is the contents of the symbol table in the form of symbol name and the type expression chain generated from the processing of the declarations in the input C program. Observe that each function results in a separate symbol table and so do the struct definitions. The dialog below shows the example program taking in C programs, and printing out the symbol table details.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o c-small-gram.o c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o c-small-lex.o c-small-lex.cc

# Building ex5 Binary
$  g++  -g  -Wall  ex5.cc  semantic_analysis.cc  c-small-gram.o  c-small-lex.o  -o  ex5

# This is a sample input source file
$ cat -n test5.c
     1  int var1;
     2  int var2[10];
     3  int var3[10][20];
     4
     5  int *var4;
     6  int *var5[50];
     7
     8
     9
    10  /* Function */
    11  int main()
    12  {
    13      var1=20;
    14  }
    15
    16  /* Another function */
    17  int ab(int one,char two, float three,int four)
    18  {
    19      /* Local Variables */
    20      int h,j;
    21      char k[56][67];
    22
    23      h=10;
    24  }
    25
```

```
# Parsing , creating and displaying Symbol table for the declarations
$ ./ex5 test5.c
Name of the Table=main Size=0

Name of the Table=ab Size=3777
Name=four Type=INTEGER Size=4 Offset=13
Name=h Type=INTEGER Size=4 Offset=17
Name=j Type=INTEGER Size=4 Offset=21
Name=k Type=ARRAY(56)->ARRAY(67)->CHAR Size=3752 Offset=25
Name=two Type=CHAR Size=1 Offset=4
Name=one Type=INTEGER Size=4 Offset=0
Name=three Type=REAL Size=8 Offset=5


# Another input source file
$ cat -n test5a.c
     1  /* Global Structure */
     2  struct my_info
     3  {
     4      int v1,v2;
     5      char c1;
     6      struct my_info *next;
     7  }  var1;
     8
     9
    10  /* Function */
    11  int main()
    12  {
    13
    14      int h;
    15
    16      h=20;
    17  }
    18

# Parsing , creating and displaying Symbol table for the declarations
$ ./ex5 test5a.c
Name of the Table=my_info Size=13
Name=next Type=POINTER->RECORD(my_info) Size=4 Offset=9
Name=v1 Type=INTEGER Size=4 Offset=0
Name=v2 Type=INTEGER Size=4 Offset=4
Name=c1 Type=CHAR Size=1 Offset=8

Name of the Table=main Size=4
Name=h Type=INTEGER Size=4 Offset=0
```

4.2.2 Type Checking

In the previous section, we saw that the processing of declarations results in filling symbol table with entries for every identifier describing the type information, scope information and other essentials. In type checking, we validate the rest of the program for usage of the identifiers in type correct manner by using these symbol table entries.

Type checking is the process of verifying, if each of the statements in the input source program respects the type system of the language and report any errors found. For example, in C language program, When we try to apply an incompatible operator like say '→' on an integer variable, we encounter a type error because it violates the type system of the language. As we saw in the introductory section, some other errors in C language program that type checking caught were assigning a float value to a char pointer, de-referencing a non-pointer variable, using the '.' operator on a non-structure variable, and so on. In type checking, we also detect some other obvious semantic errors like, say, using a 'break' statement outside a loop.

In a ***strongly typed language*** like C, none of the type errors go undetected. In a weakly typed language, the type errors can go undetected resulting in exceptions at the run time.

The type errors can be detected during compile time (as we saw in C language) or when the program is executed (as in LISP) or a combination of both. Some of the type errors can be detected at compile time, while some can only be detected during run time. For example, using a dot operator on a non-structure variable is a type error that can be detected by the compiler, while indexing an array out of bounds with an instruction like a[x] can be detected only during the run time, because it is difficult to predict the value of x at compile time with certainty.

The ***static type checking*** involves checking each and every statement in the input source program for violations of the type system ***during the compile time***. Most of the compilers also offer facility to correct some of the type errors that can occur in a program by doing type conversion or coercion. For example, consider a C language statement x = 5, where x is defined as a float variable. The compiler automatically does 'coercion' of the value 5 into 5.0 as required by the type system (since x is float), to get cleared by the semantic analyser. Contrasting this kind of 'implicit' conversions in the type by the compiler, there are also explicit conversions done by the programmer. Consider the commonly used library routine to dynamically allocate memory called 'malloc' in a C language program. This returns a pointer to void type. The returned pointer to void is typecast to the appropriate type by the programmer. This is an example of explicit conversion in type by the programmer.

The ***dynamic type checking*** involves checking every statement for type correctness ***during the execution***. The dynamic type checking requires that we evaluate every statement before execution to determine if there is any violation of the type system specified by the language. This would degrade the performance of the program. However, it offers more freedom to the programmers. LISP is a language that relies on dynamic-type checking. We shall be discussing about static-type checking in detail in this chapter.

In order to verify type correctness, A semantic analyser will have to frequently test if two type expressions represent the same type. When two type expressions represent the same type, they are called ***type equivalent***. There are a few ways type equivalence can be defined in a language.

A common way of establishing type equivalence is called the ***structural equivalence***. In structural equivalence, two type expressions are equivalent, only when both of them can be represented by the identical type expression tree. Consider the declarations in the Table 4.19. We find three variables *x*, *y* and *z* declared in it. The Type expression tree for the variables *x* and *y* corresponding to the two declarations are identical. The type expression tree of variable *z* is different from *x* or *y* as evident from Fig. 4.16. The variables *x* and *y* are structurally equivalent. C language considers two expressions to be equivalent during the type checking, if they are structurally equivalent. In the statements following the declaration in Table 4.19, we can assign *x* = *y* because they are structurally equivalent. However, we cannot assign *z* = *x* because *z* and *x* are not structurally equivalent.

Table 4.19 *Declarations*

```
char *x;
char *y;

float z ;
```

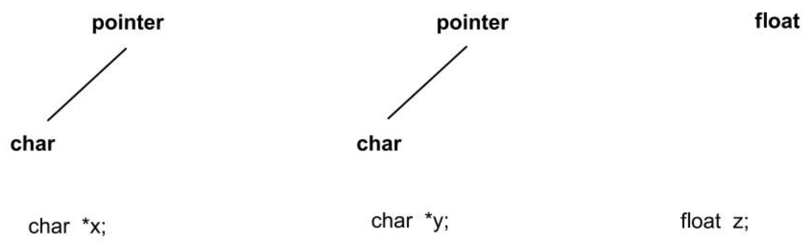


Fig. 4.16 *Type expression tree*

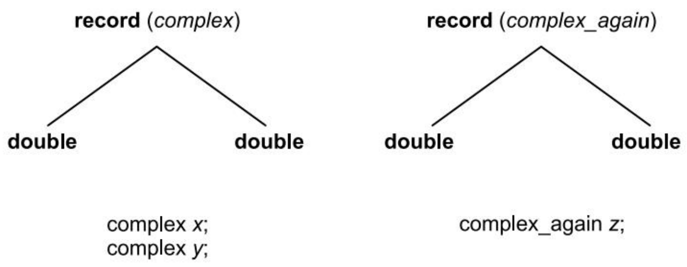
The dialog below shows two C programs given as input to GNU C compiler. The first program adheres to structural equivalence in all of the statements and the compiler successfully compiles it. The second program violates the structural equivalence in some of the statements. The compiler emits error messages indicating the lines where type equivalence is not respected.

```
# C Program using variables following structural equivalence
$ cat -n test6a.c
     1  #include <stdio.h>
     2
     3  void my_func(char *);
     4
     5  char str1[]="String 1";
     6  char str2[]="String 2";
     7
     8  int main()
     9  {
    10          char *x;
    11          char *y;
    12
    13          x = str1;
    14          y = str2;
    15
    16          /* Function my_func() can be called with parameter x or y */
    17          my_func(x);
    18          my_func(y);
    19
    20          /* x and y are type equivalent and can be assigned either way */
    21          x = y ;
    22          y = x ;
    23
    24          return(0);
    25  }
    26
    27  void my_func(char *p)
    28  {
    29          printf("%s\n",p);
    30  }
```

```
# The C Compiler compiles it successfully
$ gcc -Wall test6a.c -o test6a

# C Program using variables that do not follow structure equivalence
$ cat -n test6b.c
     1  #include <stdio.h>
     2
     3  void my_func(char *);
     4
     5  char str1[]=”String 1”;
     6  char str2[]=”String 2”;
     7
     8  int main()
     9  {
    10          char *x;
    11          float y;
    12
    13          x = str1;
    14          y = str2;
    15
    16          /* Function my_func() is being called with appropriate type */
    17          my_func(x);
    18
    19          /*
    20          Function my_func() is being called with a float
    21          as parameter instead of a pointer to char
    22          */
    23
    24          my_func(y);
    25
    26          /* x and y are not type equivalent and cannot be assigned
    27          to each other*/
    28          x = y ;
    29          y = x ;
    30
    31          return(0);
    32  }
    33
    34  void my_func(char *p)
    35  {
    36          printf(“%s\n”,p);
    37  }

# The C Compiler rejects it with appropriate error messages
$ gcc -Wall test6b.c -o test6b
test6b.c: In function ‘main’:
test6b.c:14: error: incompatible types in assignment
test6b.c:24: error: incompatible type for argument 1 of ‘my_func’
test6b.c:28: error: incompatible types in assignment
test6b.c:29: error: incompatible types in assignment
```

C language considers two expressions to be equivalent during the type checking, if they follow structural equivalence. However, there is one exception to this rule with regard to structures. The C language supports another kind of type equivalence called the ***name equivalence with regard to structures***. To understand the name equivalence, we need to be familiar with the idea of type name. Most compiled languages support the notion of a user assigning a name to the type expressions called as ***type name***. The typedef mechanism in C language is an example of user assigning a name to the type expression. The type name can then be used to define new variables. The following C language declaration shows two variables *x* and *y* belonging to the type name ‘complex’. The ‘complex’ is a type name that was defined by the user using the typedef mechanism. In name equivalence, two type expressions are equal only if they have the same type name. In the declarations below *x* and *y* are name equivalent. The variable *z* is not name equivalent with either *x* or *y* despite having identical tree representation for the data type. In the statements following the declaration, we can assign *x* = *y* because they are name equivalent. However we cannot assign *z* = *x* because *z* and *x* are not name equivalent.

```
typedef struct complex
{
      double real_part;
      double imaginary_part;
} complex;


complex x;
complex y;

typedef struct complex_again
{
      double real_part;
      double imaginary_part;
} complex_again;

complex_again z ;
```



The dialog below shows how the GNU C compiler behaves with respect to name equivalence of structures. It compiles successfully, the programs adhering to name equivalence with respect to structures. It emits error messages, when name equivalence is not respected.

```
# C Program showing Name equivalence with regard to structures
$ cat -n test6c.c
    1   #include <stdio.h>
    2
    3   typedef struct complex
    4   {
    5        double real_part;
    6        double imaginary_part;
    7   } complex;
```

```


void my_func(complex v1);

int main()
{
      complex x;
      complex y;

      y.real_part =10;
      y.imaginary_part = 20;

      /* x and y are name equivalent !!! */
      x = y ;

      /*
      my_func called with a variable of appropriate type - complex
      */
      my_func(x);

      return(0);
}

void my_func(complex v1)
{
      v1.real_part = 10;
      v1.imaginary_part = 20;
}

# The C Compiler compiles it successfully
$ gcc -Wall test6c.c -o test6c

# C Program with assignment between struct variables that are not name equivalent
$ cat -n test6d.c
#include <stdio.h>

/* Two structure definitions with Identical field types and names */
typedef struct complex
{
      double real_part;
      double imaginary_part;
} complex;

typedef struct complex_again
{
      double real_part;
      double imaginary_part;
} complex_again;


void my_func(complex v1);
```

```
  18
  19   int main()
  20   {
  21        complex x;
  22        complex_again y;
  23
  24        y.real_part =10;
  25        y.imaginary_part = 20;
  26
  27        /* x and y are NOT name equivalant !!! */
  28        x = y ;
  29
  30        /*
  31        my_func called with a variable of wrong type - complex_again
  32        */
  33        my_func(y);
  34
  35        return(0);
  36   }
  37
  38   void my_func(complex v1)
  39   {
  40        v1.real_part = 10;
  41        v1.imaginary_part = 20;
  42   }

# The C Compiler rejects it with appropriate error messages
$ gcc -Wall test6d.c -o test6d
test6d.c: In function 'main':
test6d.c:28: error: incompatible types in assignment
test6d.c:33: error: incompatible type for argument 1 of 'my_func'
```

Having understood the idea of types and type equivalence, we are now in a position to look an algorithm that establishes the type equivalence of two type expressions. Algorithm 4.6 shows a routine that can be used to check the type equivalence of two type expressions representing data types in C language. This routine can check the equivalence of type expressions using 4 basic data types, namely integer, char, real and void. It also supports the checking of type equivalence of constructed data types like record, array, pointer and function. Just as we did in the processing of the declarations in Section 4.2.1.3, we confine ourselves to simple data types without the use of any qualifiers like unsigned, long, etc.

```
int type_equal(type_expr *ty1, type_expr *ty2)
{
    type_expr *t1,*t2;
    int b1,b2;

    t1 = ty1;
    t2 = ty2;

    /* reached the end of the type chain */
    if ( (t1 == NULL ) && ( t2 == NULL ) ){
        return(TRUE);
    }

    /* One of them has terminated */
    if ( (t1 == NULL ) || ( t2 == NULL ) ){
        return(FALSE);
    }

    b1 = t1->gettype();
    b2 = t2->gettype();

    /* Not the same Kind */
    if(b1 !=b2 ){
        return(FALSE);
    }

    switch(b1)
    {
        case INTEGER:
        case CHARACTER:
        case REAL:
        case VOID_TYPE :   break;
        case RECORD :  {
                           record *r1,*r2;
                           r1 = (record *)t1;
                           r2 = (record *)t2;
                           if(r1->tagname != r2 ->tagname){ /* Name Equivalance for Structures */
                               return(FALSE);
                           }
                           break;
                       }

        case ARRAY: {
                       array *a1,*a2;
                       a1 = (array *)t1;
                       a2 = (array *)t2;

                       if(a1->limit != a2->limit){ /* Length of the Array */
                           return(FALSE);
                       }
                       break;
                    }
        case POINTER: break;
        case FUNCTION: {
                           function *f1,*f2;
                           int n,i;
                           f1 = (function *)t1;
                           f2 = (function *)t2;

                           if(f1->no_of_args != f2 -> no_of_args){ /* Number of Arguments */
                               return(FALSE);
                           }
                           n = f1->no_of_args ;
                           for(i=0;i<n;i++){
```

```
                        /* Checking Each argument for equivalence*/
                        if(!type_equal(f1->args_list[i],f2->args_list[i])){
                            return(FALSE);
                        }
                    }
                    break;
                }
        case LABEL:
        case CONSTAN:
        case TYPE_ERROR: break;
    }
    return(type_equal(t1->next,t2->next) );
```

Algorithm 4.6 *Type equivalence*

The following are the highlights of Algorithm 4.6.

- This algorithm works on the linked list representation of the type expression as illustrated in Fig. 4.14.
- The two type expressions are not equal, if they both do not belong to the same kind. The kind can be a basic data-type-like integer or a constructed data-type-like array or structure.
- Two type expressions belonging to the basic data type (e.g. integer, char, real and void) are equal if they are of the same kind. The constructed data types need to be compared for the other attributes as well. For example, two type expressions representing arrays are equal only if they match on the number of elements in the array, apart from the fact they both denote the same kind—array. In a similar manner, two type expressions representing structures (records) are equal if they have the same tag name (name equivalence), apart from the fact that they both denote the same kind—record. The type expressions representing functions are considered equal, if the number of arguments, type of each argument, and the return type are identical.
- The algorithm is recursive in nature. It checks the current node for type equivalence and then invokes itself recursively passing the next node in the type expression chain. For example, consider comparison of two Integer arrays 'a' and 'b' having 50 elements each, as shown in Fig. 4.17 along with the type expressions. The first invocation of 'type_equal' function would check the first node of a's type expression chain (array), comparing it with the first node of b's type expression chain. On the event of being successful in comparison, the 'type_equal' is invoked again recursively, this time comparing the 'int' Node.

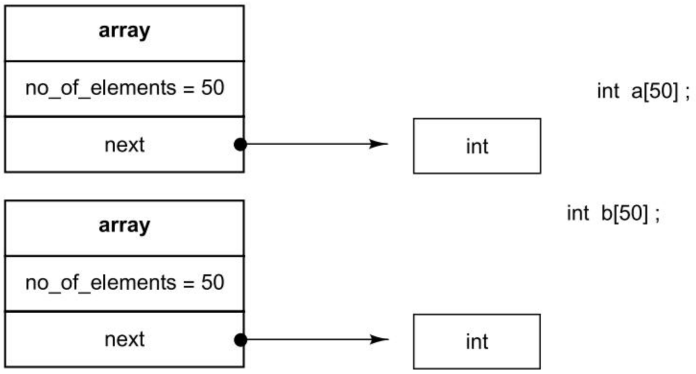


Fig. 4.17 *Type expression representing an array of 50 integers*

The algorithm to check the equivalence of two type expressions would be extensively used in the translation scheme for semantic analysis of input C language programs.

4.2.2.1 *Translation Scheme* In this section, we look at some of the productions and the semantic actions pertaining to the type checking in a semantic analyser for C language. These productions and semantic actions reinforce the ideas on type checking and other relevant concepts that we studied previously. The semantic actions are presented in a pseudo-code form. The complete implementation of a semantic analyser for C language is presented in Section 4.4.

An ‘stmt’ non-terminal is used for recognising C language statement. It takes the form of expression followed by a semicolon as given by Production 30. Table 4.20 shows ‘stmt’ and some of the other non-terminals used in the productions below and the input code strings that are generated by those non-terminals. In an expression of the form ‘unary_expr = expr’ (Production 31), the assignment is valid only when the LHS and the RHS belong to the same data type. This is checked by the type_equal( ) function. The semantic analyser reports a type error in case the data type of LHS and the RHS in the assignment statement do not match. Assigning the void data type to ‘expr’ is a means of propagating the data type to the statement. The semantic analyser reports an error in the statement if the expr is anything other than void type.

```
30  stmt   :  expr ';'
              {
                      t = expr.type
                      if( t->get_kind() != VOID ){
                           print "Semantic error in line =%d",line_no
                      }
              }
31  expr   :  unary_expr '=' expr
              {
                      t1 = unary_expr.type
                      t2 = expr.type
                      if(type_equal(t1,t2) ){
                         t = new void_type()
                      }else{
                         t = new type_error()
                      }
                      expr.type = t
              }
```

An Identifier’s type is determined by its declaration. We had seen earlier that the processing of the declaration results in a symbol table entry. In Production 32, we fetch the symbol table entry corresponding to the identifier using the lookup( ) routine. The Identifier’s type attribute is stored as a part of the symbol table entry. The semantic analyser reports an error, if the variable used is not found in the symbol table.

```
32  unary_expr   :  IDENTIFIER
                    {
                       sym_tab_entry_ptr = lookup(IDENTIFIER.name)
                       if(sym_tab_entry != NULL ){
                            t = sym_tab_entry_ptr -> type
                       }else{
                            t = new type_error();
                            printf("Variable '%s' not defined, but used in line %d
                       \n",IDENTIFIER.name,line_no);
                       }
                       unary_expr.type = t
                    }
```

A binary expression takes the form of a unary expression as given by Production 42. When two binary expressions of the integer kind are combined using an operator like '+', '–' etc, the type of the resulting binary expression is also an integer. The semantic analyser reports a type error in case any one of the combined binary expressions is not an integer. This type checking is stricter than the normal C compilers, which would allow the component binary expression to be a 'real' and perform automatic coercion to Integer. The toy compiler 'mycc' only supports Integer arithmetic.

```
38   binary_expr   :  binary_expr '*' binary_expr
                      {
                           b1 = binary_expr1.type ;
                           b2 = binary_expr2.type ;

                           k1 = b1->get_kind()
                           k2 = b2->get_kind()

                           if( ( k1 == INTEGER ) and ( k2 == INTEGER) ) {
                                t = new integer() ;
                           }else{
                                print "Incompatible Operand '*' in line=%d",line_no
                                t = new type_error()
                           }
                           binary_expr.type = t
                      }
42   binary_expr   :  unary_expr
                      {
                           unary_expr_node *u;
                           binary_expr_node *be;

                           be = new binary_expr_node();
                           u = (unary_expr_node *)$1;

                           binary_expr.type=unary_expr.type;
                      }
```

The data type of unary expression performing array access in the form of say arr[5], is given by the 'next' field of the type expression chain similar to the one we saw in Fig. 4.14. The semantic analyser in our toy compiler reports an error, if we are indexing on a non-array. Again, this is slightly different from the production C compilers that would rightly allow pointers to be indexed. The semantic analyser reports a type error in case the index of the array is not an integer.

```
43   unary_expr   :  unary_expr '[' expr ']'
                     {
                          ut = unary_expr1.type
                          kind = ut->get_kind()

                          if( kind != ARRAY ){
                               print "Incompatible Operand '[' in line=%d",line_no
                               t = new type_error()
                          }else{
                               et = expr.type
                               ekind = et->get_type()

                               if( ekind != INTEGER){
                                      print "Array index is not integer in line =%d",line_no
                                      t = new type_error()
                               }else{
                                      t = ut->next
                               }
                          }
                          unary_expr.type = t
                     }
```

For a unary expression using the address of operator in the form of say ‘&var1’, the type is given by a pointer to the type of the variable var1. This is done by adding a pointer node at the head of the type expression chain of the variable var1. The semantic analyser reports a type error in case, we are trying to use the operator ‘&’ on constants.

```
44  unary_expr  :  ‘&’ unary_expr
                   {
                        ut = unary_expr1 -> type
                        kind = ut->get_kind()

                        if( (kind == CONSTANT ) || ( kind == REAL_CONSTANT ) ){
                             printf(“Incompatible operand to & used in line %d \n”,line_no);
                             t = new type_error();
                        } else {
                             t = new pointer()
                             t->chain(unary_expr1.type)
                        }
                        unary_expr.type = t
                   }
```

The type for a unary expression using the ‘*’ operator of the form say ‘*ptr’ is given by the ‘next’ node in the type expression chain. The semantic analyser reports a type error in case the data type of ‘ptr’ is not of the kind ‘pointer’.

```
45  unary_expr  :  ‘*’ unary_expr
                   {
                        ut = unary_expr1 -> type
                        kind = ut->get_kind()

                        if( kind != POINTER ){
                             printf(“Incompatible operand to & used in line %d \n”,line_no);
                             t = new type_error();
                        } else {
                             t = ut -> next
                        }
                        unary_expr.type = t
                   }
```

The data type of unary expression performing record access in the form of say ‘myrec.field1’, is fetched from the symbol table entry pertaining to ‘field1’. The symbol table for the fields in a record is stored in the first node of the type expression chain as seen earlier in Fig. 4.14. The semantic analyser reports a type error, if the expected field name is not present in the symbol table. The unary expression for record access in the form of say ‘ptr->field1’ is also dealt similarly by fetching the type of ‘field1’ from the symbol table of the record. Again, the semantic analyser reports a type error if the expected field name is not present in the symbol table.

The data type of a unary expression performing function invocation in the form of say my_func(), is given by the return type, which is stored in the first node of the type expression tree as seen earlier in Fig. 4.14. The semantic analyser reports a type error if my_func is not of the kind ‘function’. In case of function invocation with parameters, the semantic analyser matches the data type for all the formal parameters passed in the invocation with the corresponding type for arguments recorded in the first node of the type expression chain of the function as seen in Fig. 4.14. In case there is no mismatch, the resultant unary expression is assigned the return data type of the function. In the event of mismatch, the semantic analyser reports a type error.

46	unary_expr	:	`unary_expr '.' IDENTIFIER`
			`{`
			    `ut = unary_expr1.type`
			    `kind = ut -> get_kind()`
			
			    `if(kind != RECORD ){`
			        `print "Incompatible Operator '.' used in line=%d",line_no`
			        `t = new type_error()`
			    `}else{`
			        `rt = (record *)ut              /* Record Type */`
			        `sym_tab_ptr = rt->sym_tab`
			        `field_entry = lookup(IDENTIFIER.name,sym_tab_ptr)`
			
			        `if(field_entry == NULL ){`
			            `print "Unknown Field used in line=%d",line_no`
			            `t = new type_error()`
			        `}else{`
			            `t = field_entry -> type`
			        `}`
			    `}`
			    `unary_expr.type = t`
			
			`}`

47			`unary_expr '->' IDENTIFIER`
			`{`
			    `ut = unary_expr1.type`
			    `kind = ut -> get_kind()`
			    `if(kind != POINTER ){`
			        `printf("Incompatible operator '->' used in line %d \n",line_no)`
			        `t = new type_error()`
			    `}else{`
			        `nt = ut->next`
			        `nkind = nt ->get_kind()`
			
			        `if(nkind != RECORD ){`
			            `printf("Incompatible operator '->' used in line %d`
			            `\n",line_no)`
			            `t = new type_error()`
			        `}else{`
			
			            `rt = (record *)ut          /* Record Type */`
			            `sym_tab_ptr = rt->sym_tab`
			            `field_entry = lookup(IDENTIFIER.name,sym_tab_ptr)`
			            `if(field_entry == NULL ){`
			                `print "Unknown Field used in line=%d",line_no`
			                `t = new type_error()`
			            `}else{`
			                `t = field_entry->type`
			            `}`
			        `}`
			    `}`
			    `unary_expr.type = t`
			`}`

76	unary_expr	:	unary_expr ‘(‘ ‘)’
			{
			    ut = unary_expr1.type
			    kind = ut -> get_kind()
			
			    if(kind != FUNCTION){
			        printf(“Invoking a Non Function in line %d \n”,line_no)
			        t = new type_error()
			    }else{
			        ft = (function *)ut /* Function Type */
			        if(ft->no_of_args != 0){
			            print “Expected %d arguments in line %d”,
			            unary_expr1.type.no_of_args,line_no
			            t = new type_error()
			        } else {
			            t = ut->ret_type;
			        }
			    }
			    unary_expr.type = t
			}

77	unary_expr	|	unary_expr ‘(’ args_list ‘)’
			{
			    ut = unary_expr1.type
			    kind = ut -> get_kind()
			
			    if(kind != FUNCTION){
			        printf(“Invoking a Non Function in line %d \n”,line_no)
			        t = new type_error()
			    }else{
			        ft = (function *)ut /* Function Type */
			    no_of_args = ft->no_of_args
			    mismatch_in_func_args = 0
			    for(i=0;i < no_of_args;i++){
			            if(!type_equal(ft->args[i],args_list.args[i])){
			                mismatch_in_func_args ++;
			                break;
			            }
			        }
			        if(mismatch_in_func_args == 0){
			            t = ft->ret_type;
			        }else{
			            printf(“Function Invocation used in line %d does not
			            match with its Definition/Declaration \n”,line_no);
			            t = new type_error()
			        }
			    }
			    unary_expr.type = t
			}

Table 4.20 shows some of the non-terminals and the input code strings that are generated by the non-terminal. The corresponding production used for generating the code string is also provided for getting a clear understanding.

Table 4.20 *Some non-terminals and the matching input code strings*

#	Non-terminal	Sample code strings matching the non-terminal	Production	Comments
1	stmt	x = y ;	30	
2	unary_expr	v1	32	
3	binary_expr	v1	42	
4	binary_expr	v1*v2	38	
5	expr	x = y	31	
6	expr	a = b + c	31	
7	unary_expr	arr[50]	43	
8	unary_expr	arr[index]	43	
9	unary_expr	*ptr	45	
10	unary_expr	&v1	44	
11	unary_expr	a.field1	46	
12	unary_expr	ptr->field1	47	

4.3 A SEMANTIC ANALYSER FOR C LANGUAGE

This section demonstrates the semantic analyser module of our toy C compiler (mycc) taking in sample C programs, performing semantic analysis and reporting semantic errors.

The semantic analyser implements the translation scheme that performs the semantic analysis of an input C program, parts of which were described in Sections 4.2.1.3 and 4.2.2.1. It evaluates the semantic rules during the bottom-up parsing. The semantic analyser detects the common scenarios where there is a mismatch in the types. The semantic analyser does not perform coercion among the friendly types like say float to integer. As seen in the translation scheme, the semantic analyser emits error messages depending on the semantic error in the input C program. The translation scheme is not comprehensive enough to handle all the aspects of C language. It operates on a smaller subset of the C language to bring out the ideas on the type-checking clearly.

We line up the same input C programs to our semantic analyser as we did for GNU C compiler in Example 1. It is evident from the dialog below, that our semantic analyser detects the semantic errors in those input C programs excepting for the one where there is a break statement without loop construct.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o c-small-gram.o c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
```

```
$ g++ -g -Wall -c -o c-small-lex.o c-small-lex.cc

# Building the Semantic Analyzer Binary
$ g++ -g -Wall main.cc semantic_analysis.cc c-small-gram.o c-small-lex.o -o sem_analyser

# A C Program using an undeclared variable
$ cat -n sem_err1.c
     1
     2 int main()
     3 {
     4     int a,b;
     5
     6     a=1;
     7     b=2;
     8     c=3; /* Use of undeclared variable */
     9
    10     a = b + c;
    11
    12     return(a);
    13
    14 }

# The Semantic Analyzer detects it and reports the error
$ ./sem_analyser sem_err1.c
Variable 'c' not defined, but used in line 8
Semantic error in line 8 (LHS and RHS of '=' are not the same type)
Variable 'c' not defined, but used in line 10
Incompatible operands for the operator '*' in line no =10 (Integer Operands allowed)
Semantic error in line 10 (LHS and RHS of '=' are not the same type)
Semantic Errors in the Program

# A C Program Assigning a float to char pointer
$ cat -n sem_err2.c
     1
     2 int main()
     3 {
     4     char *a;
     5
     6     float b,c;
     7
     8     b = 30.45;
     9     c = 40.36;
    10
    11     a = b + c; /* Assigning a float to char pointer */
    12
    13     return(0);
    14
    15 }

# The Semantic Analyzer detects it and reports the error
$ ./sem_analyser sem_err2.c
Incompatible operands for the operator '*' in line no =11 (Integer Operands allowed)
```

```
Semantic error in line 11 (LHS and RHS of '=' are not the same type)
Semantic Errors in the Program

# A C Program using '->' operator on a float Variable
$ cat -n sem_err3.c
     1
     2 int main()
     3 {
     4
     5     float b,c;
     6
     7     b = 30.45;
     8     c = 40.36;
     9
    10     b = c -> f1; /* using '->' operator on a float Variable */
    11
    12     return(0);
    13 }

# The Semantic Analyzer detects it and reports the error
$ ./sem_analyser sem_err3.c
Incompatible operator '->' used in line 10
Semantic error in line 10 (LHS and RHS of '=' are not the same type)
Semantic Errors in the Program

# A C Program using break statement in a non-loop context
$ cat -n sem_err4.c
     1
     2 int main()
     3 {
     4     float b,c;
     5
     6     b = 30.45;
     7     c = 40.36;
     8
     9     b = c ;
    10
    11     break; /* using break statement in a non-loop context */
    12
    13     return(0);
   14 }

# The Semantic Analyzer does NOT detect it and report error
$ ./sem_analyser sem_err4.c
Semantically correct Program

# A C Program using a float variable as function
$ cat -n sem_err5.c
     1
     2 int main()
     3 {
     4     float b,c;
     5
     6     b = 30.45;
     7     c = 40.36;
     8
     9     b = c() ; /* using a float variable as function */
    10
    11     return(0);
    12 }

# The Semantic Analyzer detects it and reports the error
$ ./sem_analyser sem_err5.c
Trying to invoke a non Function in line 9
Semantic error in line 9 (LHS and RHS of '=' are not the same type)
Semantic Errors in the Program

# A C Program declaring a variable twice in the same scope
$ cat -n sem_err6.c
     1
     2 int main()
     3 {
     4     int a;
     5
     6     int b,c;
     7     float a; /* declaring a variable twice in the same scope */
     8
     9     b = 30;
    10     c = 40;
    11
    12     a = b + c;
    13
    14     return(a);
    15
    16 }

# The Semantic Analyzer detects it and reports the error
$ ./sem_analyzer sem_err6.c
Re-declaration of the Variable 'a' in the same scope line=7
Semantic Errors in the Program
```

SUMMARY

In lexical analysis of a compiler the input program is split into tokens. In syntax analysis, the ordering of the tokens is checked to see if they formed valid syntax of the language. All the statements that are valid from syntax point of view are not legal instructions. For example, a statement in a C language program $x = y()$ is valid from syntax point of view. However, it is not a legal statement, if y is defined as an integer instead of being defined as a function. In semantic analysis, the input source is checked to see if all the statements are legal in accordance with the language rules. The semantic analysis reports the illegal statements for the user to correct them. Some of the common errors that semantic analyser reports in the case of C language input programs are shown in the dialog of the opening section.

In order to perform semantic analysis, we make use of a formalism called as ***syntax-directed translation (SDT)***. The syntax-directed translation allows us to specify and implement a semantic analyser. In syntax-directed translation, we identify the attributes of the grammar symbols in the context-free grammar and specify ***semantic rules*** relating those attributes. The semantic rules are associated with the productions. The semantic rules are evaluated to cause valuable side-effects like insertion of information into the symbol table, semantic checking, issuing of an error message, generation of intermediate code, and so on. There are 3 methods of evaluating the semantic rules in a syntax-directed definition that we discussed, namely: (1) translation interleaved with parsing, (2) parse tree method, and (3) rule-based method.

The semantic analysis consists of two main tasks: (1) processing the declarations and adding type information to the entries in the symbol table, and (2) examining the rest of the program to ensure that the identifiers are used correctly adhering to the type compatibility conventions defined by the language. This is called type checking.

We studied some of the productions and their semantic actions pertaining to declarations in the semantic analyser of C language programs in Section 4.2.1.3. We studied some of the productions and their semantic actions pertaining to type checking in the semantic analyser of C language programs in Section 4.2.2. The complete semantic analyser for C language programs was presented in Section 4.3.

REVIEW QUESTIONS AND EXERCISES

4.1 What is semantic analysis? Give some examples of errors that are detected during semantic analysis.

4.2 State whether the following statements are true or false:
   (a) A semantic analyser strips out the comments and white spaces from input source program.
   (b) Semantic analysis involves verifying if the input source forms a legal set of instructions in accordance with the language rules.
   (c) A semantic analyser checks if all the variables are declared before use.
   (d) A semantic analyser checks if the operators are used on compatible operands.

4.3 What are the main tasks involved in semantic analysis? How is it different from syntax analysis? Illustrate with an example.

4.4 What is a syntax-directed definition? What are its main characteristics? Illustrate with an example.

4.5 Explain the terms: (a) decorated parse tree, (b) dependency graph, (c) inherited and synthesised attributes. Illustrate with examples.

4.6 How do we evaluate semantic rules in a syntax-directed definition? What are the common methods used for evaluating semantic rules?

4.7 What kinds of syntax directed definitions are suited for evaluation of semantic rules during the parsing? Explain with an example.

4.8 What is a translation scheme? How is it different from a syntax-directed definition? Illustrate the order of execution of semantic actions in a translation scheme?

4.9 How do we evaluate synthesised and inherited attributes in the semantic rules during bottom-up parsing? Illustrate with an example.

4.10 How do we construct a top-down parser that can evaluate semantic rules during the parsing for an L-attributed definition? Illustrate with an example.

4.11 Describe the parse tree method of evaluating semantic rules. What are its limitations?

4.12 How do we evaluate semantic rules using the rule-based method? Describe a recursive evaluator with an example.

4.13 How do you represent a data type in a compiler? Explain an implementation approach for storing the data types in a compiler?

4.14 What is a symbol table? Explain how the symbol table in a compiler can be implemented by a hash table.

4.15 Explain some of the productions and the semantic actions pertaining to the processing of the declarations in a semantic analyser for C language.

4.16 Explain static and dynamic type checking with examples.

4.17 What is structural equivalence? Give examples of variables in C language that are structurally equivalent and structurally different.

4.18 What is name equivalence? In what context is name equivalence used during the type checking? Illustrate with an example in C language.

4.19 Write an algorithm to check the equivalence of two type expressions in C language represented by a linked list.

4.20 Explain some of the productions and the semantic actions pertaining to the type checking in a semantic analyser for C language.

5 INTERMEDIATE CODE GENERATION

Introduction

The front end of a compiler consists of lexical analysis, syntax analysis, semantic analysis and intermediate code generation. We have studied about lexical analysis, syntax analysis and semantic analysis in the previous chapters. In this chapter, we discuss about how to take the syntactically and semantically correct input source and generate intermediate code from it. The intermediate code is used by the back end of the compiler for generating the target code.

We begin the discussion by understanding the common forms of intermediate code used in compilers (Section 5.1). In Section 5.2, we take up the translation of common programming constructs in high level languages like C into intermediate code. We take a subset of the 'C' language as our reference source language and learn about the challenges associated with the translation of programming constructs like if-else, while, switch-case, etc. into intermediate code.

5.1 INTERMEDIATE FORMS

In this section, we study about the different forms of intermediate code that are commonly found in the compilers. Before we get into the details of the various forms of intermediate code that the input source can be translated into, let us first see why we need to translate the input source into an intermediate form and why not generate the final machine code itself.

Consider a monolithic compiler for C language that generates machine instructions directly from the input source for an 80×86 processor system. Let's say it needs to be modified to generate machine instructions for SPARC processor system. The effort involved in modifying the 80×86-based compiler for re-targeting to SPARC platform is high. It requires the intricate knowledge of the machine instructions of both the 80×86 system as well as SPARC System. Also, the translation to final machine code from the input source language makes the generation of optimal code difficult because it would not have the context of the entire program.

Consider another compiler that is broken into modular elements called as front end and the back end, as explained in Chapter 1. The re-targeting of such a compiler from 80×86 to SPARC system is illustrated in Fig. 5.1. The front end of the compiler for a source language remains same irrespective of the machine code generated. The output of the front end of the compiler is an intermediate form that does not depend on the specifics of the processor. The back end of the compiler converts the intermediate code into the respective machine instructions as required. This approach allows the re-use of a large portion of the compiler without modification during the re-targeting to a different processor.

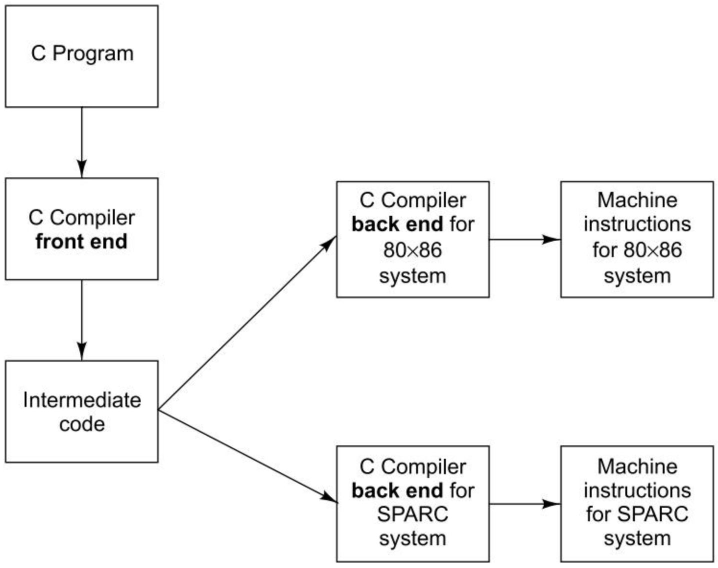

C Program
C Compiler **front end**
Intermediate code
C Compiler **back end** for 80×86 system
Machine instructions for 80×86 system
C Compiler **back end** for SPARC system
Machine instructions for SPARC system

Fig. 5.1 *Retargeting of a compiler*

Some of the advantages in this approach of breaking up the compiler into front end and back end are:

1. It is easy to re-target the compiler to generate code for newer and different processors. As seen in the discussion previously, the re-targeting of the compiler could be highly effort intensive but for the presence of intermediate code.
2. The compiler can be easily extended to support an additional input source language by adding the required front end and retaining the same back end.
3. It facilitates machine independent code optimisation. The intermediate code generated by the front end can be optimised by using several specialised techniques. This optimisation is different from the target code optimisation that can be done during the code generation for the actual processor by the back end system.

Most of the modern compilers take this approach of partitioning the job of the compiler into front end and back end.

The front end typically includes lexical analysis, syntax analysis, semantic analysis, intermediate code generation and its optimisation. The back end takes the optimised intermediate code and performs code generation for the target processor. A code optimisation specific to the processor is also performed by the backend to produce the final output.

What is intermediate code? The intermediate code is a translated form of the input source stored in some data structure like array, tree, etc. The back end generates the target code by traversing this data structure. The intermediate code is also called as ***intermediate representation (IR)***.

There are two commonly used intermediate code forms in compilers that we study in this chapter. They are:

1. Three address code (TAC)
2. Abstract syntax tree (AST)

We study about each of them in detail in the next two sections.

5.1.1 Three Address Code

The three address code (TAC) form of intermediate code consists of a list of statements called as the three address code statements. Each TAC statement is a record consisting of 4 fields, An operator, argument1, argument2 and a result. The argument1, argument2 and the result are pointers to symbol table entries pertaining to programmer defined or compiler-generated variables. The operator field holds a mnemonic for a particular operation like addition, subtraction, etc. The name—'three address code' comes from the fact that there are 3 addresses involved in each of these instructions, argument1, argument 2 and the result.

An example of a TAC statement is shown below.

Operator	Argument 1	Argument 2	Result
ADD	*y*	*z*	*x*

For the sake of readability, the arguments and the result fields are shown as name of variables, they are actually pointers to symbol table entries pertaining to those variables. In the above example, the argument 1, argument 2 and result are pointers to symbol table entries pertaining to the variables y, z and x respectively. This TAC statement represents a computation, where x is assigned the sum of y and z given textually by

$$x := y + z$$

The following shows a few more TAC statements and the computation that they represent, in textual form.

Operator	Argument 1	Argument 2	Result	
SUB	*y*	*z*	*x*	$x := y - z$
MUL	*y*	*z*	*x*	$x := y * z$
ADDR_OF	*y*	–	*x*	$x := \&y$
UMINUS	*y*	–	*x*	$x := -y$

Table 5.1 shows a couple of C language statements and their equivalent three address code statements. This gives an idea on translation from C language instructions to TAC.

Table 5.1 *Input C-statements and the translated TAC*

Input C statement	TAC statements	Comments

`a = b – c + d ;`	`_t1 := b – c`
	`_t2 := _t1 + d`
	`a := _t2 ;`
		`_t1 and _t2 are compiler generated temporaries. Note that one C statement is transformed into multiple TAC statements`

`p_new = p + ( ( p * n * r ) /100 )`	`_t1 := p * n`
	`_t2 := _t1 * r`
	`_t3 = _t2 / 100`
	`p_new = p + _t3`
		`_t1,_t2 and _t3 are compiler generated temporaries. Note that one C statement is transformed into multiple TAC statements`

The number of allowable operators (like ADD, SUB, etc.) is an important factor in the design of an intermediate representation like three address code. One end of the spectrum is a restricted operator set, which allows for easy portability to multiple architectures. A restricted feature set would mean that the front end would generate a long list of TAC instructions, forcing the optimiser and code generator to do the bulk of work. At the other end of the spectrum is a feature rich operator set in the intermediate language that allows one to take advantage of an advanced processor, but is difficult to port on to low-end processors. The usual approach is to have a minimum set of allowable operators in Intermediate language, whose equivalent machine language statements would be invariably available on any processor.

The following table shows a complete list of TAC operators that we would be using in this book.

Table 5.2 *TAC operators*

#	TAC operator	Sample TAC instruction	Textual representation	Description
1	ASSIGN	$\boxed{\text{ASSIGN}}\boxed{y}\boxed{}\boxed{x}$	$x := y$	x gets assigned the result of y op z
2	ADD	$\boxed{\text{ADD}}\boxed{y}\boxed{z}\boxed{x}$	$x := y + z$	x gets assigned the result of y added to z
3	MUL	$\boxed{\text{MUL}}\boxed{y}\boxed{z}\boxed{x}$	$x = y * z$	x gets assigned the result of y multiplied by z
4	DIV	$\boxed{\text{DIV}}\boxed{y}\boxed{z}\boxed{x}$	$x := y / z$	x gets assigned the result of y divided by z
5	SUB	$\boxed{\text{SUB}}\boxed{y}\boxed{z}\boxed{x}$	$x := y - z$	x gets assigned the result of y minus z
6	UMINUS	$\boxed{\text{UMINUS}}\boxed{y}\boxed{}\boxed{x}$	$x := -y$	x gets assigned the value of $-y$
7	L_INDEX_ASSIGN	$\boxed{\text{L\_INDEX\_ASSIGN}}\boxed{y}\boxed{i}\boxed{x}$	$x[i] := y$	$x[i]$ denotes the content of a location which is i memory units away from the pointer contained in x. $x[i]$ gets assigned the value of y.

8	R_INDEX_ASSIGN	R_INDEX_ASSIGN | *y* | *i* | *x*	$y := x[i]$	Again $x[i]$ denotes the content of a location which is *i* memory units away from the pointer contained in *x*. *y* gets assigned the value of $x[i]$.
9	ADDR_OF	ADDR_OF | *y* | | *x*	$x = \&y$	The address of the variable *y* is assigned to *x*. The operator & can be used to fetch the address of any variable, whether it is a compiler generated temporary or a user-defined variable.
10	LBL	LBL | | | my_lbl	lbl my_lbl	This marks the next TAC statement as the one with the label as 'my_lbl'.
11	GOTO	GOTO | | | my_lbl	goto my_lbl	This sample TAC statement transfers the control to the TAC statement marked as my_lbl.
12	LT	LT | *x* | *y* | my_lbl	if $x < y$ goto my_lbl	If the value of *x* is less than *y*, the control is transferred to the TAC statement marked my_lbl.
13	GT	GT | *x* | *y* | my_lbl	if $x > y$ goto my_lbl	If the value of *x* is greater than *y*, the control is transferred to the TAC statement marked my_lbl.
14	LE	LE | *x* | *y* | my_lbl	if $x <= y$ goto my_lbl	If the value of *x* is less than or equal to *y*, the control is transferred to the TAC statement marked my_lbl.
15	GE	GE | *x* | *y* | my_lbl	if $x >= y$ goto my_lbl	If the value of *x* is greater than or equal to *y*, the control is transferred to the TAC statement marked my_lbl.
16	EQ	EQ | *x* | *y* | my_lbl	if $x == y$ goto my_lbl	If the value of *x* is equal to *y*, the control is transferred to the TAC statement marked my_lbl.
17	NE	NE | *x* | *y* | my_lbl	if $x != y$ goto my_lbl	If the value of *x* is not equal to *y*, the control is transferred to the TAC statement marked my_lbl.
18	PROC_BEGIN	PROC_BEGIN | | | my_func	proc_begin my_func	This TAC statement marks the beginning of a function 'my_func'.
19	PROC_END	PROC_END | *x* | *y* | my_lbl	proc_end	This TAC statement marks the end of a function 'my_func'.

20	RETURN	RETURN | | | *x*	return *x*	Identifies *x* as the return value.
21	RETRIEVE	RETRIEVE | | |	retrieve *x*	Moves the returned value of a function into the variable *x*.
22	PARAM	PARAM | | | *x*	param *x*	*x* is identified as a parameter to a 'call' TAC statement that would follow.
23	CALL	CALL | my_func | 8 |	call my_func, 8	Transfers the control to the function my_func. The second argument is the cumulative size of all the parameters for this 'call' given in bytes.

The Table 5.3 shows some input C programs and the corresponding TAC. This gives a fair idea on most of the TAC operators mentioned above.

Table 5.3 *Input C-source and the equivalent TAC code*

Input C Source	Translated TAC	Comments

`/* Function */`
`int main()`
`{`
`  /* Local Variables */`
`  int v1,v2,v3,v4;`
 	
`  v2=200;`
`  v3=300;`
`  v4=400;`
 	
`  /* Simple assignment statements */`
`  v1 = v2 + v3 - v4 ;`
 	
`}`
	`(0)  proc_begin main`
	
	`(1)  v2 := 200`
	`(2)  v3 := 300`
	`(3)  v4 := 400`
	
	`/* v1 = v2 + v3 - v4 ; */`
	`(4)  _t0 := v2 + v3`
	`(5)  _t1 := _t0 - v4`
	`(6)  v1 := _t1`
	
	`(7)  label .L0`
	`(8)  proc_end main`
		The output TAC shows the statements using the following operators.
		• PROC_BEGIN
		• PROC_END
		• ADD
		• SUB

`/* Function */`
`int main()`
`{`
`  /* Local Variables */`
`  int var;`
`  int arr[50];`
 	
`  /* Array accesses */`
`  arr[43]=7;`
`  var = arr[43];`
`}`
	`(0)  proc_begin main`
	
	`/* arr[43]=7; */`
	`(1)  _t0 := 43 * 4`
	`(2)  _t1 := &arr`
	`(3)  _t1[_t0] := 7`
	
	`/* var = arr[43]; */`
	`(4)  _t2 := 43 * 4`
	`(5)  _t3 := &arr`
	`(6)  _t4 := _t3[_t2]`
	`(7)  var := _t4`
	
	`(8)  label .L0`
	`(9)  proc_end main`
		The output TAC shows the statements using the following operators.
		• ADDR_OF
		• L_INDEX_ASSIGN
		• R_INDEX_ASSIGN

`int x,z;`
 	
`int`
	`(0)  proc_begin func`
	
	`/* if (a < b) */`
		The output TAC shows the statements using the following operators.

```
func (int a, int b)
{
   if (a < b) {
      z = 30;
   } else {
      z = 40;
   }
   x = 90;

}
```

```
(1)  if a < b goto .L0
(2)  goto .L1

(3)  label .L0
(4)  z := 30
(5)  goto .L2
(6)  label .L1
(7)  z := 40
(8)  label .L2
(9)  x := 90
(10) label .L3
(11) proc_end func
```

- LT
- GOTO
- LBL

The TAC statements can be implemented in a compiler by having a record with 4 fields of operator, argument1, argument2 and the result. This implementation is common and is often referred to as ***quadruples*** (quads for short). The entire set of TAC statements can be represented by an array of quads.

There are subtle variants of quads approach in which the TAC statements can be implemented. They are:

1. Triples
2. Indirect Triples

In triples and indirect triples, a record with 3 fields is used to represent each of TAC statements. The 3 fields are operator, argument 1 and argument 2. In comparison with the quadruples, the result field is absent in triples. We shall see shortly how the TAC statements can be implemented with just 3 fields.

An analysis of the TAC statements (for example the ones shown in Table 5.1), indicates that the result field is mostly used in cases of compiler-generated temporaries like *_t1, _t2* etc. For assignments between programmer-defined variables, the ASSIGN TAC statement is used. The triples exploit this feature and use the statement that computes the temporary value as a reference for the temporary. For example, consider the TAC statements shown in Table 5.4.

Table 5.4 *TAC statements*

```
0 : _t1 := p * n
1 : _t2 := _t1 * r
2 : _t3 = _t2 / 100
3 : p_new = p + _t3
```

This can be represented in triples as follows.

Table 5.5 *Triples*

	Operator	Argument 1	Argument 2
0	MUL	*p*	*n*
1	MUL	(0)	*r*
2	DIV	(1)	100
3	ADD	*p*	(2)

Observe that the reference to temporaries is converted to the statements that evaluate them. For example, the temporary '*_t1*' is calculated in the triple referenced by 0. In the triple marked 1, the value of *_t1* is used. Instead of using the location *_t1* as it is done in quadruple representation, the triple that evaluated it (namely triple 0) is used as the reference in triples representation. A triple, by virtue of using this technique of referencing the statement instead of location for the temporaries has eliminated the field 'result'. It is clear that the parenthesised references are only for temporaries, while the programmer-defined variables continue using the pointers to symbol table.

Eliminating the result field in the triples notation, however, causes additional triples entries for instructions requiring 3 operands. Consider for example, an R_INDEX_ASSIGN TAC statement

$$x = y[i]$$

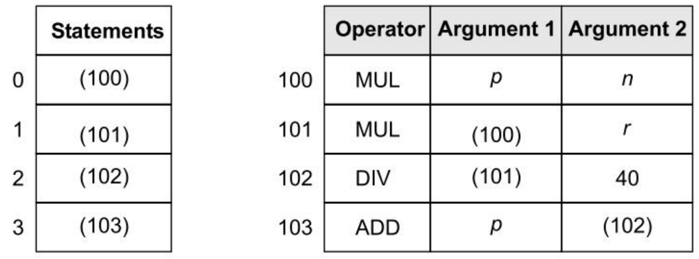
To implement this in triples, it requires two entries as shown below.

	Operator	Argument 1	Argument 2
0	R_INDEX	*y*	*i*
1	ASSIGN	*x*	(0)

The R_INDEX would be used for computing the content of the memory, which is '*i*' units away from *y*. The ASSIGN is used for assigning the result of the first computation to *x*.

In indirect triples implementation, there is a statements array that has a listing of pointers to triples in addition to the triples themselves. Every element in statements array points to one of the triples as seen in Fig. 5.2. At first sight, one might wonder about the benefit of adding an additional statements array, when the triples alone can suffice. The benefit would be evident at a later point, when the intermediate code optimisation is performed where the statements often have to be re-arranged. In indirect triples, re-arranging the statements is a matter of merely re-ordering the statement array.

The indirect triples implementation for the TAC statements in Table 5.4 is shown in Fig. 5.2.



	Statements
0	(100)
1	(101)
2	(102)
3	(103)

	Operator	Argument 1	Argument 2
100	MUL	*p*	*n*
101	MUL	(100)	*r*
102	DIV	(101)	40
103	ADD	*p*	(102)

Fig. 5.2 *Indirect triples*

Table 5.6 shows a brief comparison of the three TAC statement implementations on the parameters of indirection, suitability to optimisation and space.

Table 5.6 *Comparison of TAC implementations*

Parameter	Quadruples	Triples	Indirect Triples
Indirection	No Indirection present. All variables (temporary as well as programmer defined) have immediate access through symbol table.	Indirection present. However, since we have to allocate memory for every variable whether temporary or programmer defined, the indirection does not help much.	Indirection present.

Suitability to optimisation	The quadruples lend well to optimisation. When the statements need to be re-arranged for optimisation, the quads are moved around. For moving a quad, there is no extra dependence on other quads.	The triples do not lend well to optimisation. When the statements need to be rearranged for optimisation, the triples need to be moved. If a triple is moved, all the references (in the form of parenthesised numbers) in arg1 and arg2 arrays also have to be updated accordingly. This would be a time-consuming operation making it less compile-time efficient.	The indirect triples lend well to optimisation. When the statements need to be rearranged for optimisation, the statement list is re-ordered. The references do not change.
Space	The space required for storage is not optimum, because of the additional result field.	Requires lesser space than the quadruples.	More space required than triples. But, it can save some space as compared to quadruples because the statement list can point to the same triple for a temporary value, in case it is used more than once.

5.1.2 Abstract Syntax Tree

Some of the front-ends of compilers translate the input source into an intermediate form knows as abstract syntax tree (AST). The idea of AST can be appreciated better when it is seen in the context of parse tree.

Consider a C-statement grammar in Table 5.7 for understanding the concept of AST.

Table 5.7 *Context-free grammar*

1	c_statement	:	IDENTIFIER EQ_TO_OP c_expression SEMI_COLON
2	c_expression	:	CONSTANT
3		|	IDENTIFIER
4		|	c_expression OPERATOR CONSTANT
5		|	c_expression OPERATOR IDENTIFIER

For an input string of '*a* = *b* + *c* – 5 ;' conformant to the grammar in Table 5.7, the parse tree is shown in Fig. 5.3.

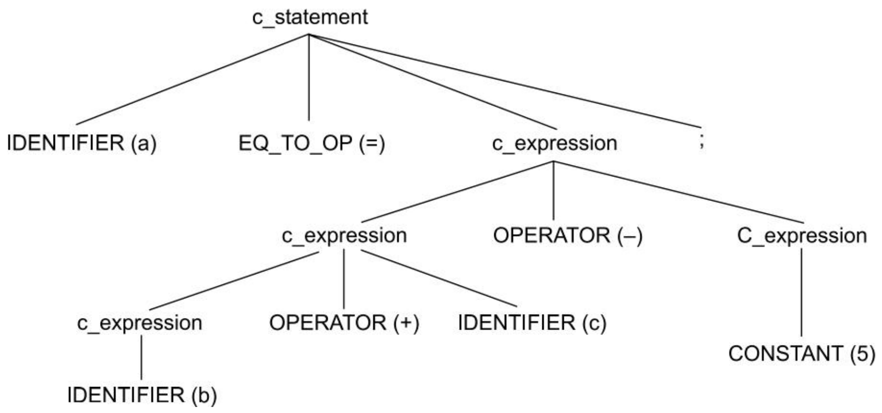


Fig. 5.3 *Parse tree for '**a** = **b** + **c** – 5;'*

The parse tree is ideal to represent the syntax of the language. There are portions of the parse tree that can be considered irrelevant from an intermediate code generation viewpoint. For example, from an intermediate code generation standpoint, it is not required to know that the IDENTIFIER was reduced to c_expression and then the 'c_expression OPERATOR CONSTANT' was reduced to c_expression and so on. The real crux in terms of intermediate code generation view comes from the leaves IDENTIFIER, OPERATOR and the CONSTANT. Another piece of information that is irrelevant from an intermediate code generation point of view is the presence of syntactic entities like semicolon, parenthesis, commas, etc.

The abstract syntax tree is a condensed version of parse tree eliminating all the syntactic sugar of the language (like semicolon, multiple levels of reduction, etc.). In AST, operands are located as leaf nodes and operators/keywords are interior nodes. The AST for the parse tree given above is shown in Fig. 5.4.

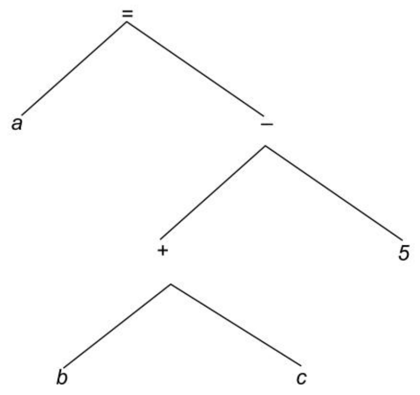


Fig. 5.4 *Abstract syntax tree for 'a = b + c – 5'*

From the AST in Fig. 5.4, we can observe that:

- The interior nodes correspond to operators like +, –, and so on.
- The leaf nodes correspond to operands like *a*, *b*, 5, etc.
- Chains of single productions like c_expression $\rightarrow$ IDENTIFIER are collapsed in comparison with the parse tree.
- The nodes of AST deal with semantic entities only. There are no AST nodes associated with syntactical entities like ';' or '(' and alike.

Another feature in AST that helps us deduce the abstract meaning of the programming construct quickly is the 'flattening' of lists like lists of declarations, lists of parameters, etc. For example, Fig. 5.5 shows an if-else construct having a block statement. The parse tree equivalent for the same block statement would typically have gone in to multiple levels owing to productions like "statement_list $\rightarrow$ statement statement_list". It is difficult to get a feel for the meaning of program by browsing the parse tree. The AST helps us deduce the meaning of the 'if' statement quickly by flattening the lists and eliminating the syntactic overhead.

The AST is usually shown in graphical form, but it is implemented in software using a record data structure. In order to implement simple assignment statements shown in Table 5.7, there are three types of AST nodes, namely operator node, identifier node and a literal node required to represent intermediate code. Each node in AST is implemented as a record with multiple fields. A ***label*** field indicating whether it is an operator or identifier or a literal is common to all the AST nodes. An identifier node has a field containing a pointer to symbol table entry of the Identifier. A literal node contains a field that stores the value of the number it represents. An operator node contains pointers to one or two of the AST nodes depending on whether it is a unary or a binary operator. These nodes are shown in Fig. 5.6.

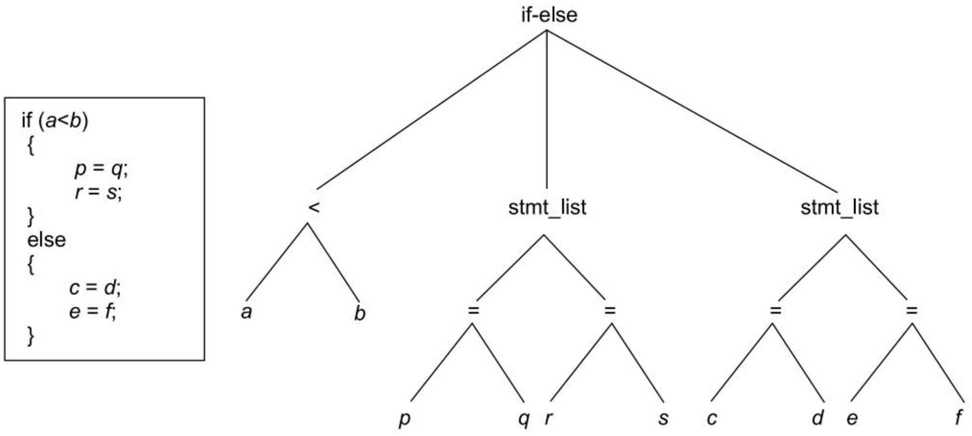


Fig. 5.5 *AST for if-else block statement construct*

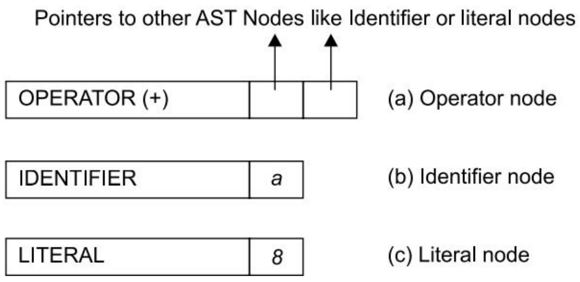


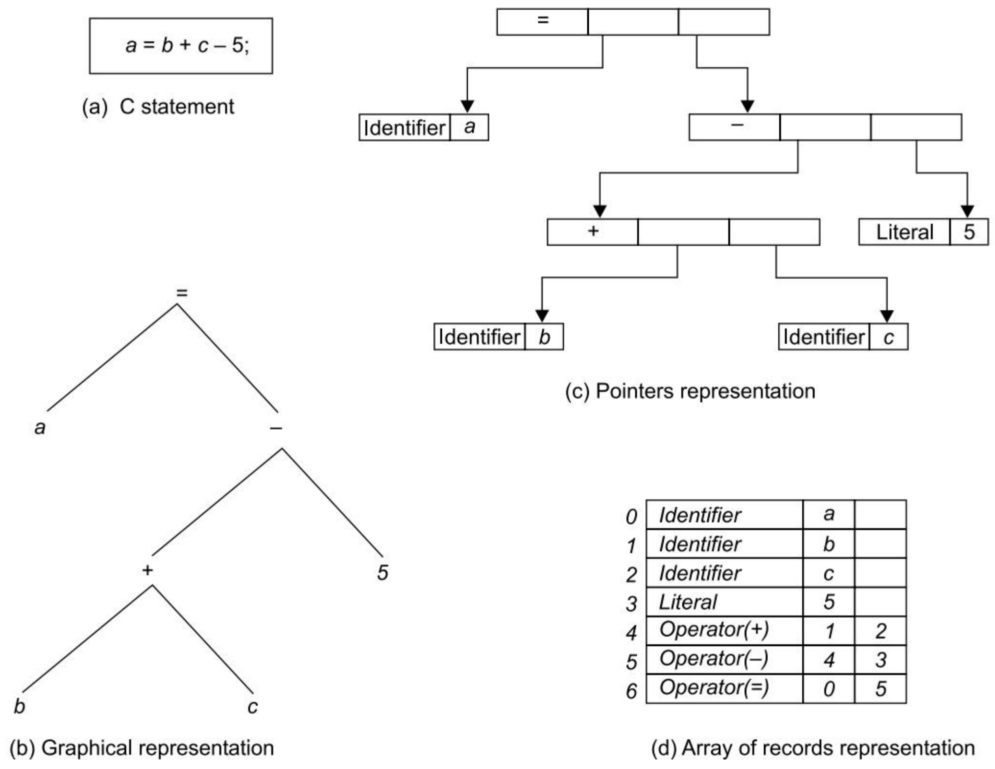
Fig. 5.6 *Nodes in AST*

An AST tree representing an assignment statement can be created by having an operator node as the root and having pointers to its children (operands) in its record structure. The entire tree can be traversed from the root node by following the fields marked for children as shown in Fig. 5.7(c), resulting in a yield of the assignment statement.

Alternatively, an AST Tree can be implemented by having an array of records of AST nodes. A new AST node is allocated from an array of records whenever required. The index to the children nodes in the fields marked for the children is stored instead of storing the pointer to children. This is shown in Fig. 5.7(d).

In either of the cases, the AST is denoted by a tree data structure. Traversal algorithms operating on the tree data structure starting from the root of the tree can enable us do specific tasks like optimisation of intermediate code, etc.

The front end of GNU compiler collection—gcc, uses AST as its intermediate code form. The AST can be dumped into a file by using the option '–fdump-translation-unit' on the command line during compilation of a C program.


a = b + c – 5;
(a) C statement
=
Identifier a
–
+
Literal 5
Identifier b
Identifier c
(c) Pointers representation
=
a
–
+
5
b
c
(b) Graphical representation
(d) Array of records representation

0	Identifier	a	
1	Identifier	b	
2	Identifier	c	
3	Literal	5	
4	Operator(+)	1	2
5	Operator(–)	4	3
6	Operator(=)	0	5

Fig. 5.7 *AST implementation*

The following are the advantages and disadvantages of using abstract syntax tree as an intermediate language.

Advantages

- The creation of an AST can be done at the time of parsing. Further analysis like type checking, optimisation of intermediate code, etc. can be easily done by operating on the AST after the parsing is completed. This gives flexibility with respect to the techniques that can be employed to have a better machine code.
- The AST lends well for intermediate code optimisation by means of reorganising code.

Disadvantages

- The AST might consume a lot of memory in order to store the hierarchical organisation of the entire program.

5.1.2.1 *Directed Acyclic Graph* An important derivative of abstract syntax tree known as ***directed acyclic graph (DAG)*** is used to reduce the amount of memory used for storing the AST tree data structure.

Consider an expression

$$k = k - 7$$

The abstract syntax tree for it is as shown in Fig. 5.8.

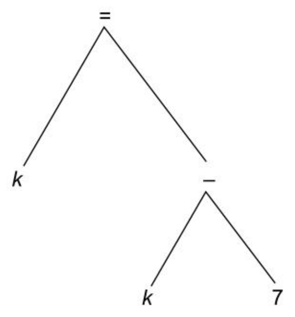


Fig. 5.8 *AST for k = k – 7*

Note that there are 2 nodes for the identifier '*k*' in Fig. 5.8, one representing *k* on the LHS of the expression and the other representing the *k* on the RHS.

The DAG identifies such common nodes and eliminates their duplication in the AST. The DAG for the expression *k* = *k* – 7 is given by

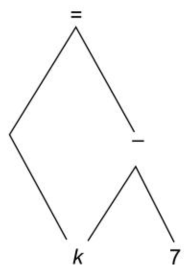


Fig. 5.9 *DAG for k = k – 7*

A DAG like the one shown in Fig. 5.9 is characterised by some of the nodes having multiple parents. In the above example, the identifier node *k* has 2 parents—the operator nodes having operator '–' and '='. The creation of DAG is identical to the AST except for the extra check to determine whether a node with identical properties already exists. In the event of the node already created before, it is chained to the existing node avoiding a duplicate node. The DAG is optimal on space as compared to the AST.

5.2 INTERMEDIATE CODE GENERATION

In this section, we study a translation scheme for translating some of the common programming constructs in higher level languages into intermediate code. A sub-set of the 'C' language is taken as the reference input source language for the discussion. The challenges associated with the translation of programming constructs like if-else, while, switch-case, etc. into intermediate code are discussed.

The discussion on the productions and semantic actions for handling each of the programming constructs is followed-up by examples showing the generation of intermediate code for the same. The intermediate code is represented in three-address code(TAC) form throughout the discussion. The bottom-up translation method is used for implementing the translation scheme and generation of intermediate code. Unless otherwise specified, all the variables used in the discussion are integers, which require 4 bytes in memory for storage each.

5.2.1 Simple Assignment Statements

The simple assignment statements are the most common statements found in programs. The assignment statements help in performing basic computations. They form the core of the programs. In this section, we learn about the translation of simple assignment statements into intermediate code.

Table 5.8 shows a few C code snippets containing simple assignment statements and the corresponding intermediate code. The variables *v*1, *v*2, etc. used in the code snippet are all assumed to be integers. The TAC statements are numbered for convenience.

Table 5.8 *Code snippets and the intermediate code*

#	Code snippet	Intermediate code

1	`v1= v2 + v3 - v4;`	`0: _t0 := v2 + v3`
		`1: _t1 := _t0 - v4`
		`2: v1 := _t1`

2	`v1 = v2 + v3 + v4 ;`	`0: _t0 := v2 + v3`
		`1: _t1 := _t0 + v4`
		`2: v1 := _t1`

3	`v1 = ( v2 – v3 ) * (v2 + 2*v3 ) ;`	`0: _t0 := v2 - v3`
		`1: _t1 := 2 * v3`
		`2: _t2 := v2 + _t1`
		`3: _t3 := _t0 * _t2`
		`4: v1 := _t3`

In the intermediate code for the simple assignment statements seen in Table 5.8, we can find that temporaries are created for storing the interim results to be later transferred to the user defined variables. For example, in the code snippet # 3, the temporary variables *_t0,_t1,_t2* and *_t3* are used for computing the interim values and later in the quad 4, a temporary value '*_t3*' is transferred to the user defined variable *v*1.

We look at some of the productions and the semantic actions pertaining to the translation of simple assignment statements in the input C language program to intermediate code. Some of the semantic actions listed below use the symbol table that is created during the processing of declarations for fetching the symbol table entry. The processing of the declarations resulting in creation of symbol table entries has been explained in the chapter on 'Semantic Analysis'. The semantic actions listed below for generating intermediate code should be treated as an extension to the actions for the semantic analysis discussed in Chapter 4.

A function 'emit' is used to generate the intermediate code in the three-address format. The function 'emit' is modelled on the quadruple representation of the three-address code. It takes four parameters, namely the TAC operator, operand1, operand2 and result, in the same order. The operator field can take one of the 23 operators shown in Table 5.2. The operand1, operand2 and result are pointers to symbol table entries holding the respective operands.

There are 4 non-terminals that are used in the translation of simple assignment statements. They are: (1) statement (2) unary expression (3) binary expression and (4) expression.

A 'stmt' non-terminal is used for recognising a C language statement. It takes the form of expression followed by a semicolon as given by Production 30.

30	stmt	:	expr ';'

A unary expression is created by reduction of lexical token IDENTIFIER. A unary expression has an attribute ‘place’. The place attribute refers to the symbol table entry of the variable where the value of unary expression is stored. When an identifier is reduced to unary expression (Production 32), the place attribute is filled with the symbol table entry of the identifier.

32	unary_expr	:	IDENTIFIER {
			 `unary_expr.place = lookup(IDENTIFIER.name); /*Symbol Table Entry*/`
			 }

The ‘place’ attribute is propagated, when a unary expression is reduced to a binary expression and also when expression with braces is reduced to a unary expression as shown by Productions 36 and 42. The place attribute is propagated from binary expression to expression via ‘or expression’ and ‘and expression’ as seen in Productions 54, 56 and 57.

36	unary_expr	:	‘(’ expr ‘)’ {
			 `unary_expr.place = expr.place`
			 }

42	binary_expr	:	unary_expr {
			 `binary_expr.place = unary_expr.place;`
			 }

54	expr	:	or_expr {
			 `expr.place = or_expr.place;`
			 }

56	or_expr	:	and_expr {
			 `or_expr.place = and_expr.place;`
			 }

57	and_expr	:	binary_expr {
			 `and_expr.place = binary_expr.place;`
			 }

In the simple assignment statement of the form unary_expr ‘=’ expr, an ASSIGN three address statement using the place attributes of expr and unary_expr is emitted.

31	expr	:	unary_expr ‘=’ expr {
			 emit(ASSIGN,expr$_1$.place,NULL,unary_expr.place);
			 }

A unary minus is handled by generating TAC statement with UMINUS operator using the function ‘emit’. A function ‘newtemp( )’ is used for creating a temporary variable like *_t0,_t1*, etc. to store the result value. The function newtemp( ) creates a symbol table entry for the temporary variable and returns it.

37	unary_expr	|	'-' unary_expr { `unary_expr.place = newtemp();` `emit(UMINUS,unary_expr`$_1$`.place,NULL,unary_expr.place);` }

The binary expression is used for storing the result of the operations involving operators like +,* etc. which require two operands. Similar to the unary expression, the main attribute for a binary expression is 'place', a pointer to the symbol table entry, where the value of binary expression is stored. The semantic actions shown for Productions 38 through 41 are along similar lines. Each of these emits TAC code using the operands and the binary operator. The result is stored in a compiler-generated temporary (e.g. _*t*1, _*t*2), created by the function 'newtemp'.

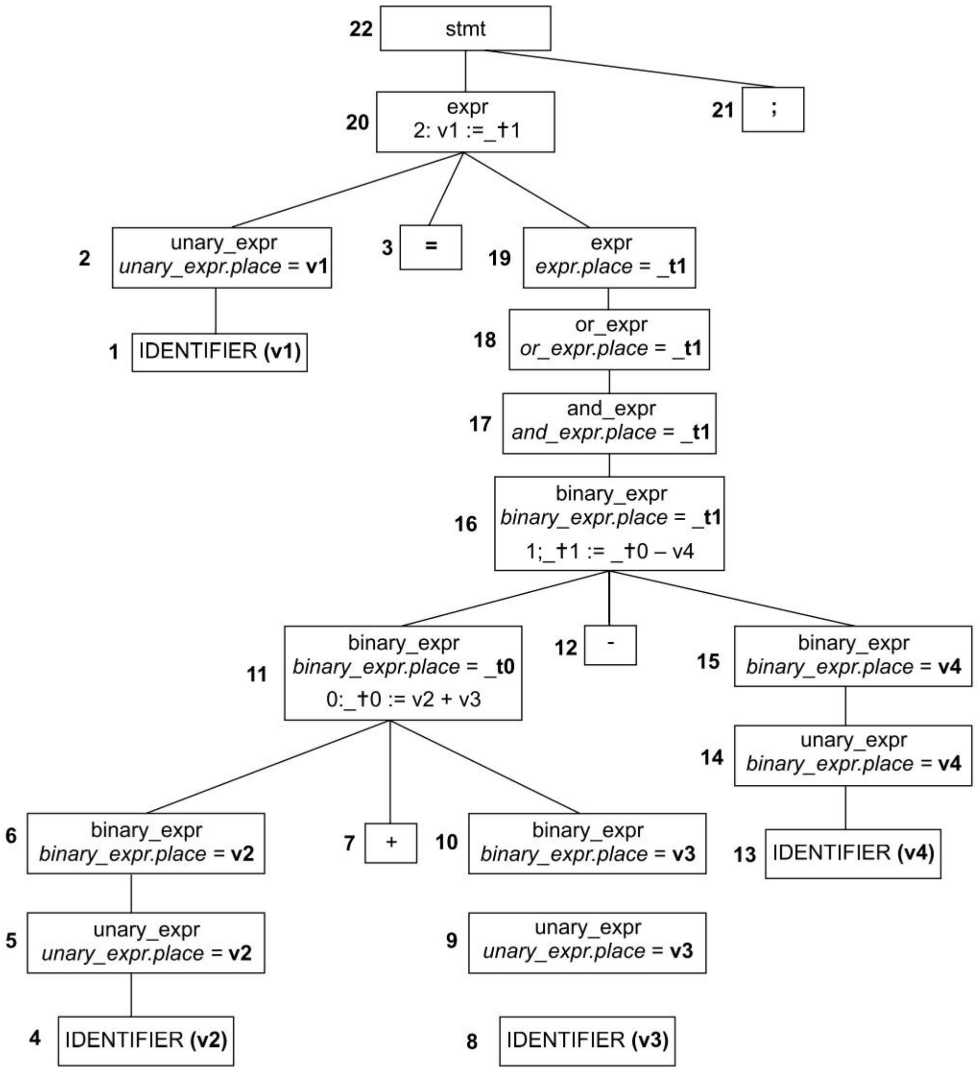
38	binary_expr	:	binary_expr '*' binary_expr { `binary_expr.place = newtemp();` `emit(MUL,binary_expr`$_1$`.place,binary_expr`$_2$`.place,binary_expr.place);` }
39		|	binary_expr '/' binary_expr { `binary_expr.place = newtemp();` `emit(DIV,binary_expr`$_1$`.place,binary_expr`$_2$`.place,binary_expr.place);` }
40		|	binary_expr '+' binary_expr { `binary_expr.place = newtemp();` `emit(PLUS,binary_expr`$_1$`.place,binary_expr`$_2$`.place,binary_expr.place);` }
41		|	binary_expr '–' binary_expr { `binary_expr.place = newtemp();` `emit(MINUS,binary_expr`$_1$`.place,binary_expr`$_2$`.place,binary_expr.place);` }

Table 5.9 shows a simple assignment statement and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.10 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation.

Table 5.9 *Input source and its translated code*

C code snippet	Translated TAC

`v1= v2+v3-v4;`	`0: _t0 := v2 + v3`
	`1: _t1 := _t0 - v4`
	`2: v1 := _t1`



22 stmt
20 expr 2: v1 :=_t1
21 ;
2 unary_expr unary_expr.place = v1
3 =
19 expr expr.place = _t1
1 IDENTIFIER (v1)
18 or_expr or_expr.place = _t1
17 and_expr and_expr.place = _t1
16 binary_expr binary_expr.place = _t1 1;_t1 := _t0 – v4
11 binary_expr binary_expr.place = _t0 0:_t0 := v2 + v3
12 -
15 binary_expr binary_expr.place = v4
14 unary_expr binary_expr.place = v4
6 binary_expr binary_expr.place = v2
7 +
10 binary_expr binary_expr.place = v3
13 IDENTIFIER (v4)
5 unary_expr unary_expr.place = v2
9 unary_expr unary_expr.place = v3
4 IDENTIFIER (v2)
8 IDENTIFIER (v3)

Fig. 5.10 *Translation of assignment statement 'v1 = v2 + v3 – v4 ;'*

5.2.2 Example 1—Simple Assignment Statements

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for simple assignment statements using the productions and semantic actions described in the preceding section. The program takes as input, a sample C input source with some simple assignment statements. The output of 'icgen' is the Intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

The PROC_BEGIN and PROC_END statements are generated at the beginning and end of a function definition respectively.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# This is an input source file
$ cat -n test1.c
     1   /* Function */
     2   int main()
     3   {
     4        /* Local Variables */
     5        int v1,v2,v3,v4;
     6
     7        v2=200;
     8        v3=300;
     9        v4=400;
    10
    11        /* Simple assignment statements */
    12        v1 = v2 + v3 - v4 ;
    13
    14    }

# Generating Intermediate code for simple assignment statements
$ ./icgen test1.c
(0) proc_begin main
(1) v2 := 200
(2) v3 := 300
(3) v4 := 400
(4) _t0 := v2 + v3
(5) _t1 := _t0 - v4
(6) v1 := _t1
(7) label .L0
(8) proc_end main

# Another input source file
$ cat -n test1a.c
     1   /* Function */
     2   int main()
     3   {
     4       /* Local Variables */
     5       int v1,v2,v3,v4;
     6
     7       /* Simple assignment statements */
     8       v1=50;
     9       v2=30;
    10
```

```
          v3 = v1 + v2 ;
          v4 = v1 + 34 - (45 *v3 - v2 ) ;

    }

# Generating IC for assignment statements
$ ./icgen test1a.c
(0) proc_begin main
(1) v1 := 50
(2) v2 := 30
(3) _t0 := v1 + v2
(4) v3 := _t0
(5) _t1 := v1 + 34
(6) _t2 := 45 * v3
(7) _t3 := _t2 - v2
(8) _t4 := _t1 - _t3
(9) v4 := _t4
(10) label .L0
(11) proc_end main
```

5.2.3 Arrays

One of the important data structures used in higher level programming languages is arrays. The C language supports single and multi-dimensional arrays. In this section, we learn about generation of intermediate code for statements using array references in the input source.

Arrays are stored in memory as a block of contiguous locations. An element in an array can be accessed by computing the offset from the base of the array and fetching the value in that location. Consider an array stored at an address '*base*' and the width of the array element is given by '*w*'. The memory layout for the array is shown in Fig. 5.11.

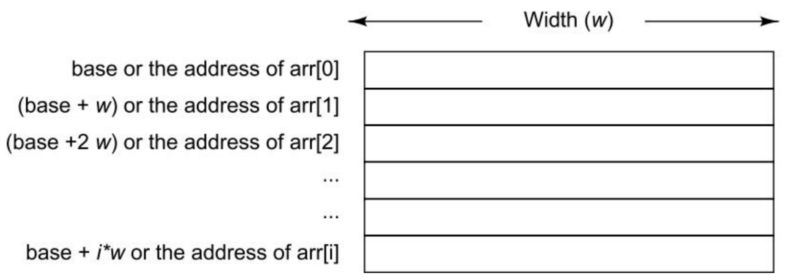

Width (w)
base or the address of arr[0]
(base + w) or the address of arr[1]
(base +2 w) or the address of arr[2]
...
...
base + i*w or the address of arr[i]

Fig. 5.11 *Array memory layout*

From Fig. 5.11, it is evident that the address of '*i*'th element in the array is given by 'base + *i***w*'.

The TAC generated for an array access of '*i*'th element, uses the 'base' or starting address of the array and an *offset* given by '*i***w*' to index to the correct location. Table 5.10 shows a couple of code snippets making array references and the corresponding intermediate code. We can see from Table 5.10 that there are two types of TAC statements that are useful in dealing with array references. They are the address assignment statement and the indexed assignment statement given in Table 5.2.

Table 5.10 *TAC statements for array references*

#	Code snippet	Intermediate code
1	`var = arr[i];`	`0: _t0 := i * 4        /* Assuming the width of integer is 4 Bytes */`
		`1: _t1 := &arr         /* Fetching the base address */`
		`2: _t2 := _t1[_t0]     /* _t1 contains 'base' and _t0 contains 'offset' */`
		`3: var := _t2`
2	`var = arr[43];`	`0: _t0 := 43 * 4       /* Assuming the width of integer is 4 Bytes */`
		`1: _t1 := &arr         /* Fetching the base address */`
		`2: _t2 := _t1[_t0]     /* _t1 contains 'base' and _t0 contains 'offset' */`
		`3: var := _t2`

A multi-dimensional array can be stored in memory in row major form or column major form. In row major form the elements of the multi-dimensional array are stored row by row, while the column major form stores elements of the multi-dimensional array column by column. Figure 5.12 shows the memory layout of storage in row major form representing a two-dimensional array of size arr[2][3]. Figure 5.13 shows the memory layout of storage in column major form for the same array.

First Row	arr[0][0]
	arr[0][1]
	arr[0][2]
Second Row	arr[1][0]
	arr[1][1]
	arr[1][2]

Fig. 5.12 *Row major form of storage for multi-dimensional array (arr[2][3])*

First Column	arr[0][0]
	arr[1][0]
Second Column	arr[0][1]
	arr[1][1]
Third Column	arr[0][2]
	arr[1][2]

Fig. 5.13 *Column major form of storage for multi-dimensional array (arr[2][3])*

Consider an array of size arr[$d1$][$d2$], with each element of width w. From the Fig. 5.12, it is evident that in the row major form, the address corresponding to the element a[$i1$][$i2$] is given by 'base + ($i1$*$d2$ + $i2$)*w'. From the Fig. 5.13, it is evident that in the column major form, the address corresponding to the element a[$i1$][$i2$] is given by 'base + ($i1$ + $i2$*$d1$)*w'. The 'C' language compilers store the arrays in row major form. In the following discussion, we assume row major form of storage for multi-dimensional arrays.

For a generic multi-dimensional array of 'n' dimensions, arr[d_1][d_2][d_3]......[d_n], the address of an element arr[i_1][i_2][i_3]......[i_n] using row major form given by

$$\boxed{\text{base} + (i_1 * d_2 * d_3 * d_4 \ldots\ldots d_n * w) + (i_2 * d_3 * d_4 * d_5 \ldots\ldots d_n * w) + (i_3 * d_4 * d_5 * d_6 \ldots\ldots d_n * w) \ldots. + (i_n * w)} \quad \ldots. \text{ Equation 5.1}$$

Let us examine Equation 5.1 carefully in terms of how each index i_n contributes to the offset calculation. The contributing factor in Equation 5.1 corresponding to the first index i_1 is given by $i_1 * d_2 * d_3 * d_4 \ldots\ldots d_n * w$. Similarly, the next index i_2, contributes $i_2 * d_3 * d_4 * d_5 \ldots\ldots d_n * w$. The index i_3 contributes $i_3 * d_4 * d_5 \ldots\ldots d_n * w$, and so on. The index i_n, contributes $i_n * w$.

In order to translate array references, there are two attributes, ***'place' and 'offset'*** associated with unary expression. The 'place' would typically hold the base address of the array and the 'offset' holds the sum of parenthesised expressions in Equation 5.1. When the array index i_1 is processed during an access for arr[i_1][i_2][i_3]......[i_n], in an input source, the contributing factor corresponding to it, i.e. ($i_1 * d_2 * d_3 * d_4 \ldots\ldots d_n * w$) is evaluated and held in the attribute 'offset'. Similarly, when the index i_2 is processed the corresponding factor ($i_2 * d_3 * d_4 * d_5 \ldots\ldots d_n * w$) is computed and summed up with the value already in offset. This goes on until all the indexes till i_n are processed. Let's take an example to clarify these ideas.

Consider an array usage as shown in Listing 5.1. The line 4 declares the array and in line 9 an element in the array is accessed.

```
1  int main()
2  {
3        int h;
4        int arr[30][40][50];
5
6        h=30;
7
8        /* Array accesses */
9        arr[6][23][9]=h;
10
11  }
```

Listing 5.1 *Array access in program*

After the declaration on line 4 is processed in semantic analysis, a type expression chain as shown in Fig. 5.14 is set up.

int arr [30][40][50];

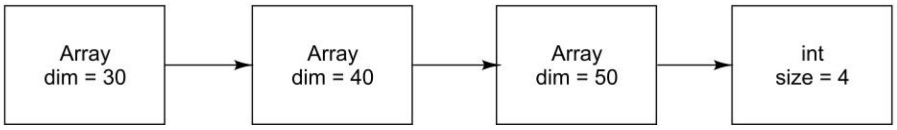


Fig. 5.14 *Type expression chain*

During the processing of array access on line 9, i.e. arr[6][23][9], when the first index arr[6] is parsed, the factor $i_1 * d_2 * d_3 * d_4 \ldots\ldots d_n * w$ is evaluated according to Equation 5.1. The value of $d_2 * d_3 * w$ can be readily calculated by following the type expression chain starting from the next dimension till the end as 40*50*4, which is 8000. The contributing factor from the first index [6] is thus 6*8000. Similarly, when the next index, which is [23] is parsed, the value of $d_3 * w$ can be calculated by following the type expression chain starting from the next dimension till the end as 50*4, which is 200. The contributing factor for the second index [23] is thus 23*200. When the final index, which is [9] is parsed, the contributing factor, computed from the type expression chain is 9*4. Thus for the access arr[6][23][9], the memory location is the base of the array, arr plus an offset, which is the sum of the contributing factors calculated above. The TAC for the same is shown below.

Table 5.11 *TAC statements for array reference arr[6][23][9] = h*

```
0: _t0 := 6 * 8000    /* contributing factor from the first index 6 */
1: _t1 := &arr
2: _t2 := 23 * 200    /* contributing factor from the second index 23 */
3: _t0 := _t2 + _t0
4: _t3 := 9 * 4       /* contributing factor from the third index 9 */
5: _t0 := _t3 + _t0
6: _t1[_t0] := h
```

The ideas just discussed are embodied in the productions and the semantic actions shown below. In order to support array references, the semantic actions for some of the productions seen in Section 5.3.1 require modification. The reader can appreciate the fact that if the attribute ‘offset’ of the unary expression is NULL, it would make this translation similar to the one shown in Section 5.3.1.

When a unary expression is created from IDENTIFIER (Production 32) or expression (Production 36), the offset attribute is initialised with NULL. The offset attribute would be filled in later, when array access is found in Production 43.

32	unary_expr	:	IDENTIFIER {
			`/* Symbol Table Entry */`
			`unary_expr.place = lookup(IDENTIFIER.name);`
			`unary_expr.offset = NULL ;`
			}

36	unary_expr	:	‘(’ expr ‘)’ {
			`unary_expr.place = expr.place`
			`unary_expr.offset = NULL ;`
			}

The array references are handled by Production 43 shown below. In the semantic action, an ADDR_OF statement is emitted, when the first index is sighted. The first index can be identified by the condition of offset being NULL. The value of offset is determined by calculating the contributing factor as given in Equation 5.1. For calculating the contributing factor, the index given by ‘expr.place’ and a couple of functions getsize( ) and make_lit_tab_entry( ) are used. The getsize( ) function calculates the factor $d_{(i+1)} *d_{(i+2)} *d_{(i+3)}.....d_n * w$ for an index ‘i’. The function make_lit_tab_entry( ) makes an entry in the literal table for literals like say 1, 2, etc. Observe that Production 43 is left recursive. This helps in fetching one index at a time, when multi-dimensional access is made.

43	unary_expr	:	unary_expr '[' expr ']' {
			`sz_of_unit= getsize(unary_expr`$_1$`.expr_type.next);`
			`se_ptr = make_lit_tab_entry(sz_of_unit);`
			`t = newtemp();`
			`emit(MUL,expr.place,se_ptr,t);`
			`if(unary_expr`$_1$`.offset == NULL ){`
			`  t1 = newtemp();`
			`  emit(ADDR_OF,unary_expr`$_1$`.place,NULL,t`$_1$`);`
			`  unary_expr.place = t1;`
			`  unary_expr.offset = t;`
			`}else{`
			`  emit(PLUS,t,unary_expr`$_1$`.offset,unary_expr.offset);`
			`}`
			}

When a unary expression is used, the content of the unary expression is accessed by using the attributes, place and offset as given by Production 42 and 31.

42	binary_expr	|	unary_expr {
			`if(unary_expr.offset == NULL ){`
			`    binary_expr.place = unary_expr.place;`
			`}else{`
			`    binary_expr.place = newtemp();`
			`    emit(R_INDEX_ASSIGN,unary_expr.place,unary_expr. offset,`
			`                             binary_expr.place);`
			`}`
			}
31	expr	:	unary_expr '=' expr {
			`if(unary_expr.offset == NULL ){`
			`    emit(ASSIGN,expr`$_1$`.place,NULL,unary_expr.place);`
			`}else{`
			`    emit(L_INDEX_ASSIGN,unary_expr.offset,expr`$_1$`. place, unary_expr.place);`
			`}`
			}

Table 5.12 shows an array access statement and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.15 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation.

Table 5.12 *Input source and its translated code*

C code snippet	Translated TAC	Comments
`x = arr[5][9];`	`0: _t0 := 5 * 120`	The array 'arr' is assumed to be declared as *int arr[20][30] ;*
	`1: _t1 := &arr`	
	`2: _t2 := 9 * 4`	The value 120 in quad 0 comes from 30 * 4, where 30 is the next dimension and 4 is the size of integer.
	`3: _t0 := _t2 + _t0`	
	`4: _t3 := _t1[_t0]`	
	`5: x := _t3`	

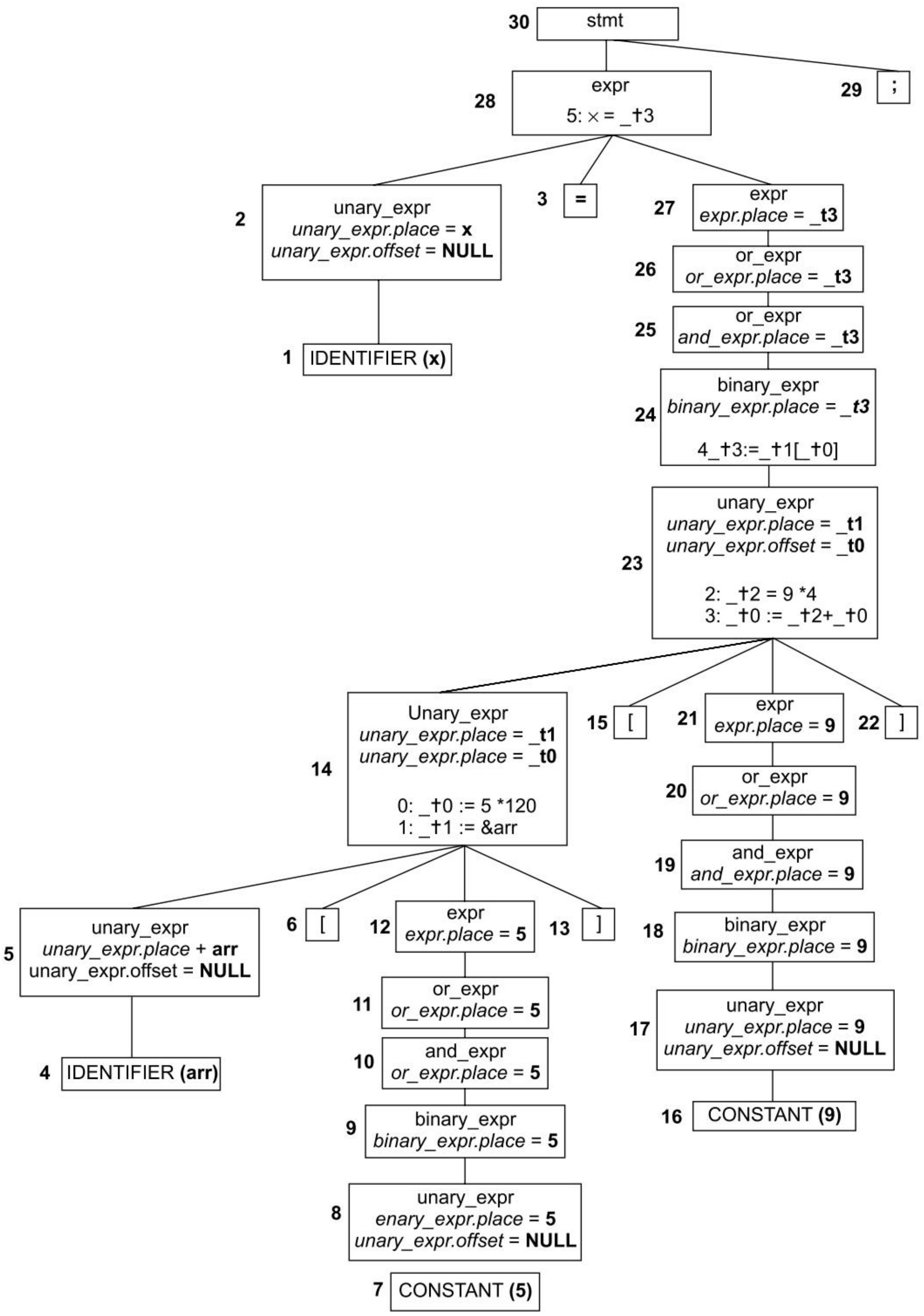


Fig. 5.15 *Translation of statement x = arr[5][9]*

5.2.4 Example 2—Array Access

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for array references using the productions and semantic actions described in the preceding section. The program takes as input, a sample C input source with some array references. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# This is a input source file
$ cat -n test2.c
    1
    2  /* Function */
    3  int main()
    4  {
    5      /* Local Variables */
    6      int var;
    7      int arr[50];
    8
    9      /* Array accesses */
   10      arr[43]=7;
   11      var = arr[43];
   12  }

# Generating Intermediate code for array access statements
$ ./icgen test2.c
(0) proc_begin main
(1) _t0 := 43 * 4
(2) _t1 := &arr
(3) _t1[_t0] := 7
(4) _t2 := 43 * 4
(5) _t3 := &arr
(6) _t4 := _t3[_t2]
(7) var := _t4
(8) label .L0
(9) proc_end main


# Another input source file
$ cat -n test2a.c
```

```
     1  /* Variables */
     2  int x,arr[20][30];
     3
     4
     5  /* Function */
     6  int main()
     7  {
     8      /* Array accesses */
     9      x = arr[5][9];
    10  }

# Generating IC for Array access
$ ./icgen test2a.c
(0) proc_begin main
(1) _t0 := 5 * 120
(2) _t1 := &arr
(3) _t2 := 9 * 4
(4) _t0 := _t2 + _t0
(5) _t3 := _t1[_t0]
(6) x := _t3
(7) label .L0
(8) proc_end main

# Another input source file
$ cat -n test2b.c
     1  /* Function */
     2  int main()
     3  {
     4      /* Local Variables */
     5      int h;
     6      int arr[30][40][50];
     7
     8      /* Initialization */
     9      h=30;
    10
    11      /* Array accesses */
    12      arr[6][23][9]=h;
    13  }

# Generating IC for Array access
$ ./icgen test2b.c
(0) proc_begin main
(1) h := 30
(2) _t0 := 6 * 8000
(3) _t1 := &arr
(4) _t2 := 23 * 200
(5) _t0 := _t2 + _t0
(6) _t3 := 9 * 4
(7) _t0 := _t3 + _t0
(8) _t1[_t0] := h
(9) label .L0
(10) proc_end main
```

5.2.5 Pointers and Address Operators

Pointers are an important part of higher level programming languages. They are used to set up data structures like linked lists, trees, and so on. In this section, we learn about translating the C language statements involving pointers into intermediate code in three address format.

The statements in C language involving pointers consists of the usage of two operators namely '*' and '&'. In a later section, we will also look at the '–>' operator, which de-references an element in a structure. Table 5.13 shows some C code snippets using the '*' and '&' operators and the corresponding intermediate code. This gives an idea on how the intermediate code looks like for accesses using pointer and address of operators. We can see from Table 5.13 that the left-indexed assign and right-indexed assign TAC operators are used in the translation of the '*' operator. The '&' operator in C input source is translated to the ADDR_OF operator in the TAC statements.

Table 5.13 *C code snippets and the intermediate code*

#	Sample C code snippet	Intermediate code	Comments

1	`p = &x ;`
	`*p = 10 ;`
		`0: _t0 := &x`
		`1: p := _t0`
		`2: p[0] := 10`
			`x is assumed to be an integer variable.`

2	`p=&x;`
	`y=*p;`
		`0: _t0 := &x`
		`1: p := _t0`
		`2: _t1 := p[0]`
		`3: y := _t1`
			`x is assumed to be an integer variable.`

3	`p = &arr[3]`
	`*p = 10 ;`
		`0: _t0 := 3 * 4`
		`1: _t1 := &arr`
		`2: _t2 := _t1 + _t0`
		`3: p := _t2`
		`4: p[0] := 10`
			`The array 'arr' is assumed to be declared as` *`int arr[20];`*
			
			`The size of an integer is 4 Bytes.`

4	`p = &arr[3]`
	`y=*p;`
		`0: _t2 := 3 * 4`
		`1: _t3 := &arr`
		`2: _t4 := _t3 + _t2`
		`3: p := _t4`
		`4: _t5 := p[0]`
		`5: x := _t5`
			`The array 'arr' is assumed to be declared as` *`int arr[20];`*
			
			`The size of an integer is 4 Bytes.`

We have seen in the previous section that the translation of access to any variable in the input program is done by using two attributes of unary expression namely the 'place' and 'offset'. When an array access is translated, the place contains the base address of the variable and the offset contains the number of memory units from the base for that particular access. When a simple variable access in the input source program is translated, the offset attribute is NULL. The same attributes place, and offset are used to handle the two operators '*' and '&'.

The translation of '& unary_expr' in the input source program (Production 44) involves fetching the address of the unary expression. For simple variables, the address is given by place attribute alone, since the offset would be NULL. The ADDR_OF TAC statement is emitted as a part of the translation. For an array variable, which is identified by 'place' and 'offset' attributes, the address is simply a sum of the place and offset attribute.

```
44  unary_expr  :  '&' unary_expr {
                          t= newtemp()
                          if(unary_expr1.offset == NULL ){
                               emit(ADDR_OF,unary_expr1.place,NULL,t);
                          }else{
                               emit(PLUS,unary_expr1.place,unary_expr1.offset,t);
                          }
                          unary_expr.place=t;
                          unary_expr.offset = 0;
                   }
```

The translation of '* unary_expr' in the input source program (Production 45) involves fetching the content of the unary expression. For simple variables, the content is obtained by getting the content at 'place' attribute alone, since the offset would be NULL. For an array variable, which is identified by 'place' and 'offset' attributes, the content is given by 'place' attribute indexed with 'offset' attribute using the R_INDEX_ASSIGN statement shown in Table 5.2.

```
45  unary_expr  :  '*' unary_expr  {

                          t=newtemp();

                          if(unary_expr1.offset == NULL ){
                               unary_expr.place=unary_expr1.place;
                               unary_expr.offset=make_lit_tbl_entry(0);
                          }else{
                               t1=newtemp();
                               emit(R_INDEX_ASSIGN,
                                              unary_expr1.place,
                                              unary_expr1.offset,t1);
                               unary_expr.place=t1;
                               unary_expr.offset=make_lit_tbl_entry(0);
                          }


                   }
```

Table 5.14 shows statement using '&' operator and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.16 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom up translation.

Table 5.14 *Input source and its translated code*

C code snippet	Translated TAC	Comments

`x = &arr[3];`	`0: _t0 := 3 * 4`
	`1: _t1 := &arr`
	`2: _t2 := _t1 + _t0`
	`3: x := _t2`
		The array 'arr' is assumed to be declared as *int arr[20] ;*
		
		The size of an integer is 4 bytes.

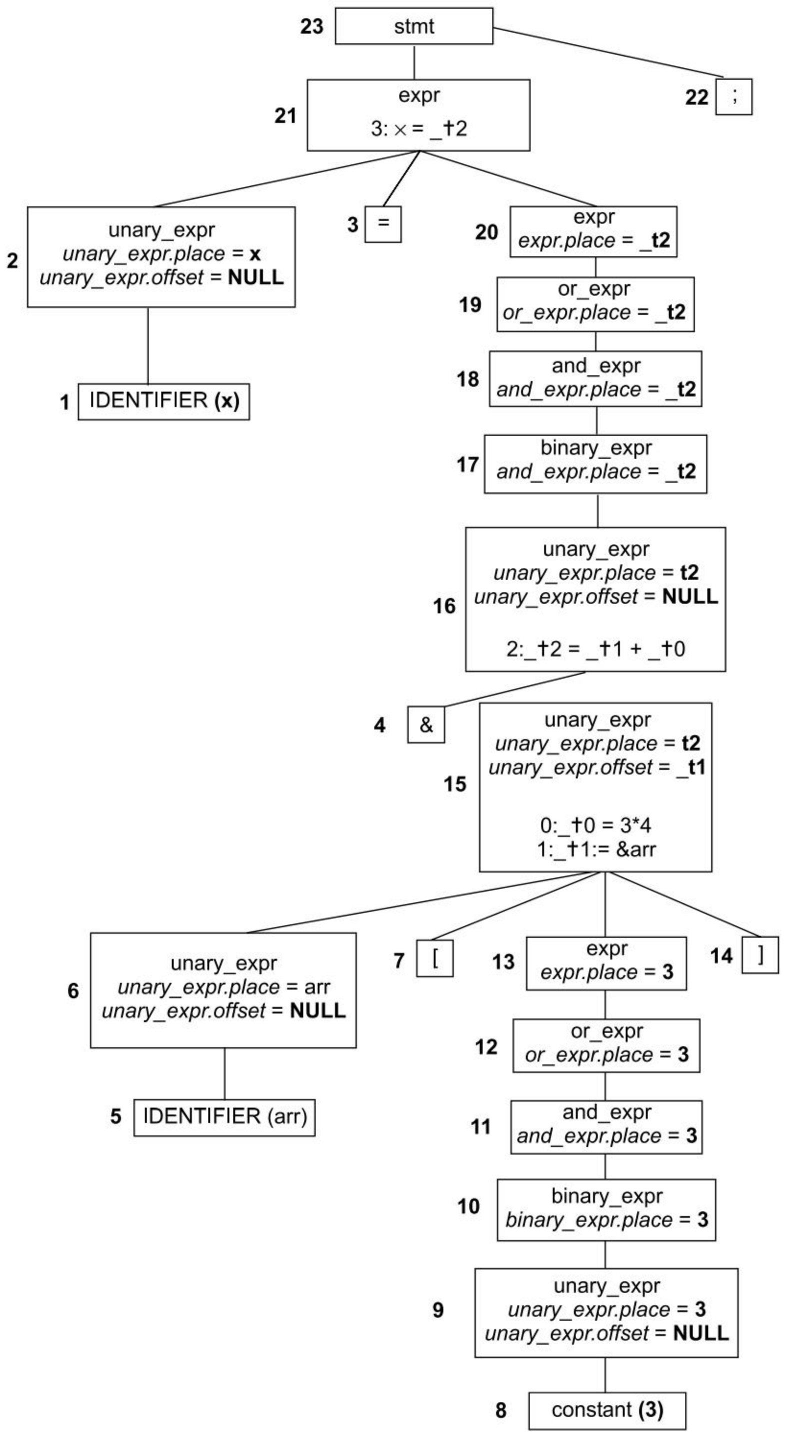


Fig. 5.16 *Translation of a statement 'x = &arr[3] ; '*

5.2.6 Example 3—Pointers and Address Operators

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving '*' and '&' operators using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom up translation method. The program takes as input, a sample C input source with some statements using '*' and '&' operators. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# This is an input source file
$ cat -n test3.c
     1   int *p;
     2   int x;
     3
     4   /* Function */
     5   int main()
     6   {
     7        /* Move 10 into x */
     8        p=&x;
     9        *p=10;
    10   }

# Generating IC for statements with pointer and Address operators
$ ./icgen test3.c
(0) proc_begin main
(1) _t0 := &x
(2) p := _t0
(3) p[0] := 10
(4) label .L0
(5) proc_end main

# Input source file
$ cat -n test3a.c
     1   int *p;
     2   int x,y;
     3
     4   /* Function */
     5   int main()
     6   {
     7         x=10;
     8
     9         /* Move value of x into y */
    10         p=&x;
    11         y=*p;
    12   }

# Generating IC
$ ./icgen test3a.c
(0) proc_begin main
(1) x := 10
(2) _t0 := &x
(3) p := _t0
(4) _t1 := p[0]
(5) y := _t1
(6) label .L0
(7) proc_end main

# Input source file
$ cat -n test3b.c
     1   int *p;
     2   int arr[10];
     3
     4   /* Function */
     5   int main()
     6   {
     7         /* Move 10 into arr[3] */
     8         p=&arr[3];
     9         *p=10;
    10   }

# Generating IC
$ ./icgen test3b.c
(0) proc_begin main
(1) _t0 := 3 * 4
(2) _t1 := &arr
(3) _t2 := _t1 + _t0
(4) p := _t2
(5) p[0] := 10
(6) label .L0
(7) proc_end main

# Input source file
$ cat -n test3c.c
     1   int *p;
     2   int arr[10];
     3   int x;
     4
     5   /* Function */
     6   int main()
     7   {
     8         arr[3]=10;
     9
    10         /* Move 10 into x */
```

```
            p=&arr[3];
            x=*p;
     }

# Generating IC
$ ./icgen test3c.c
(0) proc_begin main
(1) _t0 := 3 * 4
(2) _t1 := &arr
(3) _t1[_t0] := 10
(4) _t2 := 3 * 4
(5) _t3 := &arr
(6) _t4 := _t3 + _t2
(7) p := _t4
(8) _t5 := p[0]
(9) x := _t5
(10) label .L0
(11) proc_end main
```

5.2.7 Record Access

A structure or record is a common facility used in higher-level languages like C. A record can consist of multiple fields of different data types. In Chapter 4, on ‘Semantic Analysis’, we saw that a separate symbol table was created for each structure declaration. The fields of the structure are contained as elements of that symbol table. When C statement containing the structure references is translated, the symbol table is used to calculate the offset required to access a particular member of the structure.

Consider the C code snippet containing the usage of a structure in Table 5.15 for understanding the important aspects of intermediate code generation for a structure reference. The code snippet shows a structure ‘my_data’ containing two integer fields student_id and age. In C language, there are two ways of accessing a field in a structure in a C statement, one by using the ‘.’ operator and the other by using ‘->’ operator. Table 5.15 shows the usage of the dot (.) operator for accessing a field in a structure.

Table 5.15 *C code snippet using dot operator*

```
..
..
struct my_data
{
        int student_id;
        int age;
}d1;

..
..

d1.age=20;
..
..
```

Table 5.16 shows the intermediate code for the code snippet in Table 5.15. The struct references are translated to indexed assign TAC statements. The base address of the structure is used as the base in the indexed operation. The offset in bytes for the field of reference, which is ‘age’ in this case, is used an index.

The field ‘age’ follows the field ‘student_id’ which is 4 bytes. Hence, the offset of the field ‘age’ is 4 bytes from the base address.

Table 5.16 *TAC statements for the struct access by dot operator*

```
0: _t1 := &d1
1: _t2 := 4
2: _t1[_t2] := 20
```

Consider the C code snippet in Table 5.17 containing the usage of ‘→’ (arrow) operator for accessing structure fields. A variable ‘*ptr*’ accesses the fields in the structure by using the → operator.

Table 5.17 *C code snippet using arrow operator*

```
..
..
struct my_data
{
     int student_id;
     int age;
}*ptr;

..
..

ptr->age=20;
..
..
```

Table 5.18 shows the intermediate code for the code snippet in Table 5.17. As in the previous case, the struct references are translated to indexed assign TAC statements. The value of the ‘*ptr*’ is used as the base in the indexed operation. The offset in bytes for the field of reference, which is ‘age’ in this case, is used an index. The field ‘age’ follows the field ‘student_id’, which is 4 bytes. Hence the offset of the field ‘age’ is 4 bytes from the base address.

Table 5.18 *TAC statements for the struct access by arrow operator*

```
0: _t3 := ptr
1: _t4 := 4
2: _t3[_t4] := 20
```

The translation of structure references (Productions 46 and 47 below) in the statements involves handling the two operators, ‘.’ and ‘→’. For the translation of structure reference, when the dot operator is used, the following steps are involved: (a) Fetch the address of the unary expression preceding the dot (b) Calculate the offset of the field following the dot and (c) Use the values in (a) & (b) in the indexed TAC statement to access the memory associated with the field. Similarly, when a ‘→’ operator is used to access a field in a structure, the steps involved are: (a) Fetch the content of the unary expression preceding the ‘→’ (b) Calculate the offset of the field following the ‘→’ and (c) Use the values in (a) & (b) in the indexed TAC statement to access the memory associated with the field. The calculation of offset of the field in both the cases is done by using the symbol table pertaining to the structure definition, which holds the offsets of all the fields associated with the structure. The semantic actions of Productions 46 and 47 also show some of the sanity checks to be performed (e.g. checking if the field name exists in the structure, etc.) before emitting the intermediate code.

```
46  unary_expr  :  unary_expr '.' IDENTIFIER {

                      type = unary_expr1.expr_type

                      if(type != RECORD ){
                          Error Msg("Type Mismatch");
                          exit();
                      }

                      sym_tab_ptr = type.fields

                      sym_tab_entry_ptr = sym_tab_ptr.find(IDENTIFIER.name)

                      if(sym_tab_entry_ptr == NULL ){
                          Error Msg("Accessing a non-existent field");
                          exit();
                      }

                      offset = sym_tab_entry_ptr.offset ;
                      unary_expr.expr_type= sym_tab_entry_ptr.type;

                      if(unary_expr1.offset == NULL ){
                          t= newtemp()
                          emit(ADDR_OF,unary_expr1.place,NULL,t);
                          unary_expr.place = t;
                          unary_expr.offset = newtemp();
                          emit(ASSIGN,offset,NULL,unary_expr.offset);
                      }else{
                          emit(PLUS,unary_expr1.offset,offset,unary_expr1.offset);
                          unary_expr.place = unary_expr1.place ;
                          unary_expr.offset = unary_expr1.offset
                      }

                   }

47             |  unary_expr PTR_OP IDENTIFIER {

                      type = unary_expr1.expr_type

                      if( (type != POINTER) || (type.next != RECORD ) ){
                          Error Msg("Type Mismatch");
                          exit();

                      }

                      sym_tab_ptr = type.next.fields

                      sym_tab_entry_ptr = sym_tab_ptr.find(IDENTIFIER.name)

                      if(sym_tab_entry_ptr == NULL ){
                          Error Msg("Accessing a non-existent field");
                          exit();
                      }


                      offset = sym_tab_entry_ptr.offset ;
                      unary_expr.expr_type= sym_tab_entry_ptr.type;

                      t = newtemp();
```

```
    if(unary_expr1.offset == NULL ){
        emit(ASSIGN,unary_expr1.place,NULL,t);
    }else{
        emit(R_INDEX_ASSIGN,unary_expr1.place,unary_expr1.offset,t);
    }

    unary_expr.place = t;
    unary_expr.offset = newtemp();

    emit(ASSIGN,offset,NULL,unary_expr.offset);
}
```

Table 5.19 shows a structure access using '.' operator and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.17 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom up translation. The offset of the field 'age' is assumed to be 4 bytes.

Table 5.19 *Input source and its translated code*

C code snippet	Translated TAC

`x.age=3 ;`	`0: _t0 := &x`
	`1: _t1 := 4`
	`2: _t0[_t1] := 3`

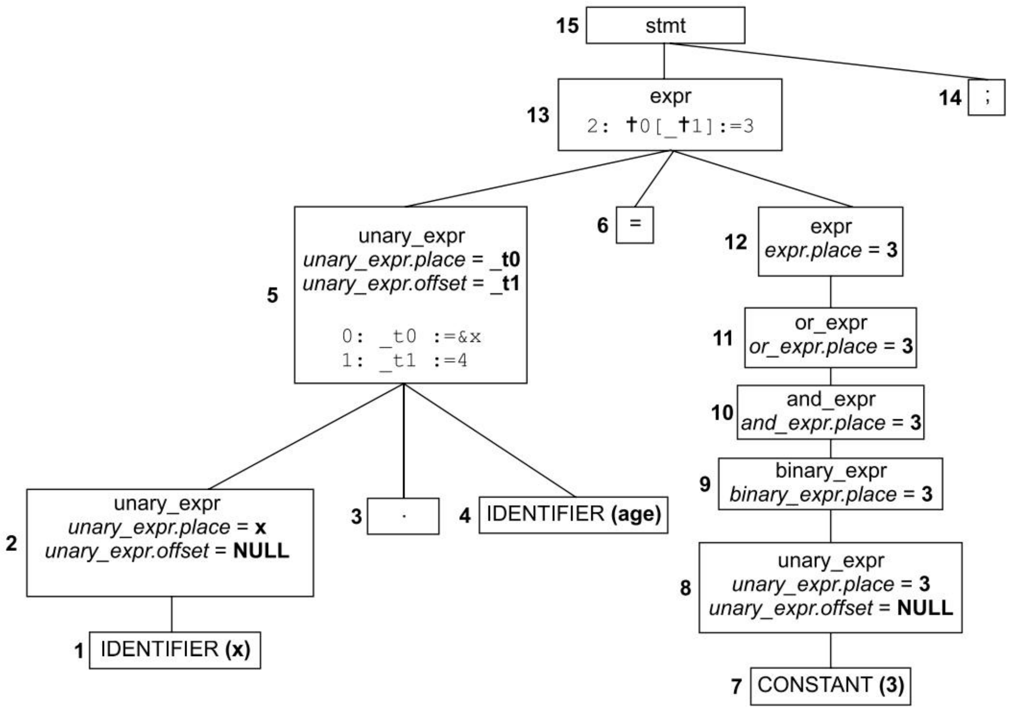

15 stmt
13 expr 2: _t0[_t1]:=3
14 ;
5 unary_expr unary_expr.place = _t0 unary_expr.offset = _t1 0: _t0 :=&x 1: _t1 :=4
6 =
12 expr expr.place = 3
11 or_expr or_expr.place = 3
10 and_expr and_expr.place = 3
9 binary_expr binary_expr.place = 3
8 unary_expr unary_expr.place = 3 unary_expr.offset = NULL
7 CONSTANT (3)
2 unary_expr unary_expr.place = x unary_expr.offset = NULL
3 .
4 IDENTIFIER (age)
1 IDENTIFIER (x)

Fig. 5.17 *The translation of statement 'x.age = 3 ;'*

Table 5.20 shows a structure access using '→' operator and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.18 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom up translation. The offset of the field 'age' is assumed to be 4 bytes.

Table 5.20 *Input source and its translated code*

C code snippet	Translated TAC

`ptr->age=3 ;`	`0: _t0 := ptr`
	`1: _t1 := 4`
	`2: _t0[_t1] := 3`

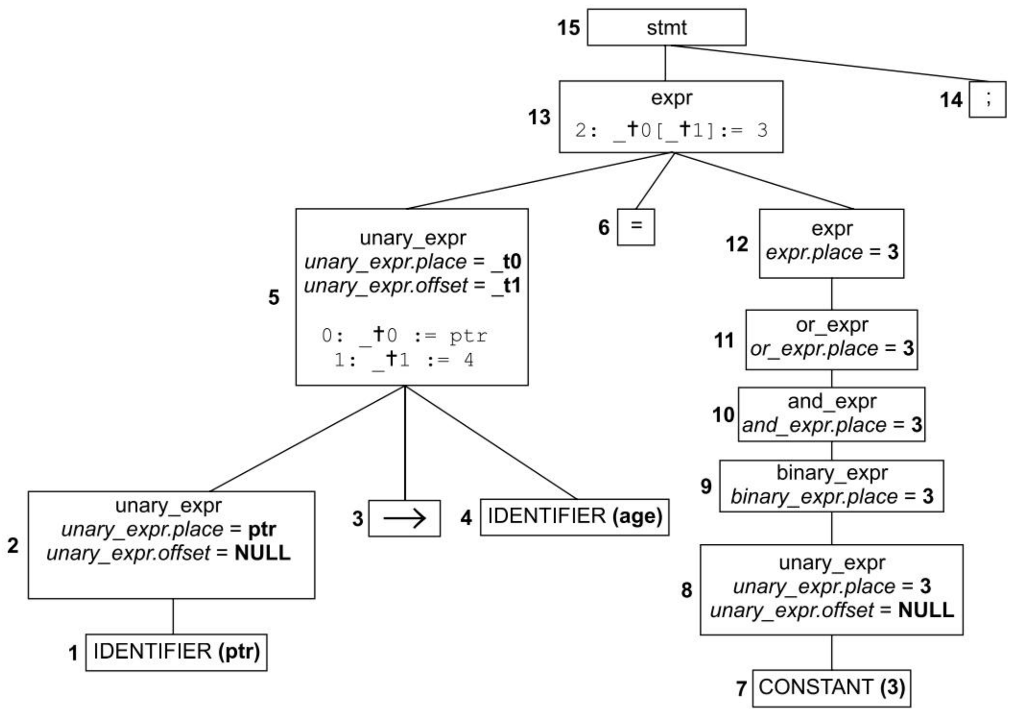


Fig. 5.18 *The translation of statement 'ptr→age = 3 ;'*

5.2.8 Example 4—Translation of Record References

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving structure references using the productions and semantic actions described in the preceding section. The program takes as input, a sample C input source with some statements involving structure references. The output of 'icgen' is the intermediate code in TAC format generated from processing the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++   -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# This is an input source file
$ cat -n test4.c
     1  struct my_data
     2  {
     3      int student_id;
     4      int age;
     5  }d1;
     6
     7
     8  /* Function */
     9  int main()
    10  {
    11      struct my_data *ptr;
    12
    13      ptr = &d1;
    14
    15      /* Access by '.' operator */
    16      d1.age=20;
    17
    18      /* Access by '->' operator */
    19      ptr->age=20;
    20
    21 }

# Generating IC
$ ./icgen test4.c
 (0) proc_begin main
 (1) _t0 := &d1
 (2) ptr := _t0
 (3) _t1 := &d1
 (4) _t2 := 4
 (5) _t1[_t2] := 20
 (6) _t3 := ptr
 (7) _t4 := 4
 (8) _t3[_t4] := 20
 (9) label .L0
(10) proc_end main

# Input source file
$ cat -n test4a.c
    1  struct dob{
    2      int day;
    3      int mon;
    4      int year;
    5  }ab;
    6
    7  struct my_data
    8  {
    9      int age;
   10      int student_id;
   11      struct dob student_dob;
   12  }d1;
   13
   14
   15  /* Function */
   16  int main()
   17  {
   18      /* Local Variables */
   19      int h,j,k;
   20
   21      /* Initialisation */
   22      d1.age=20;
   23      d1.student_id=4567;
   24      d1.student_dob.day=17;
   25      d1.student_dob.mon=11;
   26      d1.student_dob.year=1967;
   27  }

# Generating IC
$ ./icgen test4a.c
 (0) proc_begin main
 (1) _t0 := &d1
 (2) _t1 := 0
 (3) _t0[_t1] := 20
 (4) _t2 := &d1
 (5) _t3 := 4
 (6) _t2[_t3] := 4567
 (7) _t4 := &d1
 (8) _t5 := 8
 (9) _t5 := _t5 + 0
(10) _t4[_t5] := 17
(11) _t6 := &d1
(12) _t7 := 8
(13) _t7 := _t7 + 4
```

```
(14) _t6[_t7] := 11
(15) _t8 := &d1
(16) _t9 := 8
(17) _t9 := _t9 + 8
(18) _t8[_t9] := 1967
(19) label .L0
(20) proc_end main

# Input source file
$ cat -n test4b.c
     1  struct dob{
     2      int day;
     3      int mon;
     4      int year;
     5  }ab;
     6
     7  struct my_data
     8  {
     9      int age;
    10      int student_id;
    11      struct dob student_dob;
    12 }d1;

    13
    14
    15  /* Function */
    16  int main()
    17  {
    18      /* Local Variables */
    19      struct my_data *ptr;
    20
    21      /* Initialisation */
    22      ptr=&d1;
    23
    24      ptr->age=20;
    25      ptr->student_id=4567;
    26
    27      ptr->student_dob.mon=11;
    28  }

# Generating IC
$ ./icgen test4b.c
 (0) proc_begin main
 (1) _t0 := &d1
 (2) ptr := _t0
 (3) _t1 := ptr
 (4) _t2 := 0
 (5) _t1[_t2] := 20
 (6) _t3 := ptr
 (7) _t4 := 4
 (8) _t3[_t4] := 4567
 (9) _t5 := ptr
(10) _t6 := 8
(11) _t6 := _t6 + 4
(12) _t5[_t6] := 11
(13) label .L0
(14) proc_end main
```

5.2.9 If-else Statements

In this section, we learn about the translation of simple flow of control statements namely, the 'if' and 'if-else' statements. These statements are used in higher-level languages to make decisions and execute the designated code.

Consider the C code snippet containing an 'if-else' statement in Table 5.21 for understanding the important aspects of intermediate code generation for if-else statement. The variables used in the code snippet are all assumed to be integers.

Table 5.21 *A C code snippet with if-else condition*

```
..
..

if (a < b) {
   z = 30;
} else {
   z = 40;
}
x=90;

..
..
```

The expression that is being evaluated in the 'if' condition namely ($a < b$) is called as a ***Boolean test expression*** or simply ***test expression***. It can have a result of TRUE or FALSE. In the code snippet shown above, if the Boolean test expression ($a < b$) is TRUE, then the control flows to the code where z is assigned 30. If the Boolean test expression is false, then the control flows into the code where z is assigned 40.

A simple Boolean test expression like the one shown in Table 5.21, i.e. ($a < b$) can also use other relation operators for the comparison like greater than (>), less than or equal to (<=), etc.

Programming languages also allow for compound Boolean expressions formed out of logical combination of simpler Boolean expressions using operators known as ***Boolean operators***. The common Boolean operators are ***and (&&)***, ***or (||)*** operators. An example of a compound Boolean expression formed by using the OR Boolean operator is shown in Table 5.22.

Table 5.22 *A C code snippet showing compound Boolean expression*

```
..
..
if ( (a < b ) || ( c < d ) )
{
     z = 30;
}
else
{
     z = 40;
}
..
..
```

The compound Boolean expression ($a < b$) || ($c < d$) is TRUE in situation where a is less than b or c is less than d. The value 30 is assigned to z, when the compound Boolean expression is TRUE. The value 40 is assigned to z when the compound Boolean expression is FALSE.

Table 5.23 below shows the intermediate code for the if-else statement shown in Table 5.21. The TAC statements are numbered for easy reference.

Table 5.23 *Translated intermediate code*

```
0: if a < b goto .L0
1: goto .L1
2: label .L0
3: z := 30
4: goto .L2
5: label .L1
6: z := 40
7: label .L2
8: x := 90
```

The first two TAC statements, numbered 0 and 1 correspond to the decision-making based on Boolean expression ($a < b$). The flow of control reaches the statement numbered 2 with the label. L0 only if the Boolean expression ($a < b$) is true. The statement numbered 5 with label. L1 is reached only if $a < b$ is false. This method of construction of the intermediate code where the position reached by the program signifies the result of the Boolean expression is known as ***flow of control*** method of translating Boolean expressions. This method of translating the Boolean expressions is used in the rest of the chapter.

Is it possible to generate intermediate code for a Boolean expression in one pass during the parsing by having a translation scheme?

The main challenge in implementing such a translation scheme is that the target labels for the jump statements are not known at the time of generating code for the test expression. To illustrate the point, consider the code snippet in Table 5.21. At the time of reducing the Boolean expression $a < b$ and generating the TAC statement for it, there is no information about the target label to jump. The statement $z = 30$, where it needs to jump on the condition being true, is not yet processed for syntax analysis. Also, the target label to jump when the condition is false, i.e. $z = 40$ is not yet processed for syntax analysis at the time of generating TAC instructions for the Boolean expression ($a < b$). The target label to jump when the Boolean expression is true or false becomes clear only after processing several statements following the Boolean expression.

Fortunately, there is a technique by which this issue can be overcome to generate intermediate code for Boolean expression in a single pass during the syntax analysis itself. Let's understand this technique with an example.

Consider the generation of intermediate code for the code snippet in Table 5.23. At the time of reducing the Boolean expression $a < b$, the intermediate code is generated by keeping the target label unfilled. The generated intermediate code is stored in an array called ***quads***. The generated TAC instructions are shown below. The numbering given to the each of the TAC instructions can also be used as an index of the respective quad in the quads array. For example, the instruction $z := 40$ is available at the index 6 in the quads array.

```
0 : if a<b goto __
1 : goto __

2 : lbl .L0
3 : z := 30
4 : goto __

5 : lbl .L1
6 : z := 40
7 : lbl .L2
```

During the intermediate code generation, few data structures are updated to remember the indexes in the quads array, having unfilled target labels, i.e. 0, 1 and 4. At a later stage, when the parser has completely reduced the 'if-else' statement, the quads array is revisited to plug in the unfilled target labels as shown below.

```
0 : if a<b goto .L0
1 : goto .L1

2 : lbl .L0
3 : z := 30
4 : goto .L2

5 : lbl .L1
6 : z := 40
7 : lbl .L2
```

This approach of generating TAC statements with unfilled target labels and then at a later stage filling up the target labels when the information is available is called as ***backpatching***. This technique can be used to generate intermediate code for flow of control statements using Boolean expressions like the if-else, while, switch-case statements, and so on.

Let's briefly touch upon some of the data structures and functions that are used to implement backpatching.

An array called ***'quads'*** array is used to hold the sequence of Quadruples generated. Any TAC statement in the quads array can be accessed using its index. For example, in the quads array containing the following TAC statements.

```
0 : if a<b goto __
1 : goto __

2 : lbl .L0
3 : z := 30
4 : goto __

5 : lbl .L1
6 : z := 40
7 : lbl .L2
```

The quads[3] contains the TAC statement '*z* := 30'. The quads[4] contains the TAC statement with an unfilled target label 'goto _' and so on.

The TAC statements are populated in the quads array in a sequence using the '*emit*' function. The first call to emit function would fill in a TAC statement at quads[0], the next call to emit would store the TAC statement at quads[1] and so on. A variable *nextquad* is used to maintain the index in the quads array where the next emit statement would be stored. As expected, this would be incremented each time an emit function call is made.

To get an idea on the other common functions required during backpatching, consider the following input source.

```
if  ( (a < b ) || ( c < d ) )
{
        i=j ;
}
l=m;
```

The following TAC statements are generated in the quads array during the processing of the compound Boolean expression ($(a < b) \|\| (c < d)$).

```
0 : if a < b goto __
1 : goto .L0

2 : label .L0
3 : if c < d goto __
4 : goto __
```

When the expression $a < b$ is true or when $c < d$ is true, the control goes to the same label. In other words, the target label for the TAC statements at index 0 and 3 are the same. It is efficient to make a list of the labels that need the same target label, so that they can be filled in at one shot. A list of quads called say *'truelist'*, indicative of target label to be jumped when the compound Boolean expression is *true*, is created with the elements 0 and 3. The truelist is given by the list {0,3}. In a similar way, the *'falselist'* for the compound Boolean expression is given by list {4}. The list creation and appending elements into it are carried out by the following functions.

- **makelist(quad_no).** This creates a list with the quad_no as the only element in a list. It returns the pointer to the list so created.
- **quadlist_merge(newlist, list1,list2).** This concatenates the items in list1 and list2 and puts them in newlist.

Backpatching is accomplished by a function *backpatch(list, label)*. This fills the unfilled goto's target labels in the list with the label. For example, using the *truelist* {0, 3} mentioned above, an invocation of the function backpatch (truelist,.L1) would fill in the target label .L1 in the list of statements given by truelist, i.e. in quads at index 0 and 3 as shown below.

```
0 : if a < b goto .L1
1 : goto .L0

2 : label .L0
3 : if c < d goto .L1
4 : goto __

5 : label .L1
6 : i := j
7 : label .L2
8 : l := m
9 : label .L3
```

Armed with the functions to perform backpatching and some of the theory we studied above, let's study about some of the productions and their semantic actions for generating TAC statements for if and if-else statements.

The processing of binary expression (Productions 59–60) involves emitting the TAC with the appropriate conditional check like GT, LT, etc. followed by an unconditional jump statement. The conditional jump statement would be added to the 'truelist' of the binary expression, since that is the statement it would jump when the binary expression is true. The unconditional jump statement is added to the false list, since the target is the statement to be jumped when the binary expression is false. The handling of other relation operators like '<=','>=', etc. are similar in nature. The attributes of truelist and falselist are relayed on from binary_expr to test_expr (production 53) through and_expr,or_expr and expr.

59	binary_expr	:	binary_expr ‘<’ binary_expr { binary_expr.truelist = makelist(next_quad); binary_expr.falselist = makelist((next_quad+1)); emit(LT,binary_expr$_1$.place,binary_expr$_2$.place,NULL); emit (GOTO,NULL,NULL,NULL); }
60		|	binary_expr ‘>’ binary_expr { binary_expr.truelist = makelist(next_quad); binary_expr.falselist = makelist((next_quad+1)); emit(GT,binary_expr$_1$.place,binary_expr$_2$.place,NULL); emit (GOTO,NULL,NULL,NULL); }
53	test_expr	:	expr { test_expr.place = expr.place; test_expr.truelist = expr.truelist; test_expr.falselist = expr.falselist; }
54	expr	:	or_expr { expr.place = or_expr.place; expr.truelist = or_expr.truelist; expr.falselist = or_expr.falselist; }
56	or_expr	:	and_expr { or_expr.place = and_expr.place; or_expr.truelist = and_expr.truelist; or_expr.falselist = and_expr.falselist; }
57	and_expr	:	binary_expr { and_expr.place = binary_expr.place; and_expr.truelist = binary_expr.truelist; and_expr.falselist = binary_expr.falselist; }

The OR_OP (||) is used to combine two binary expressions with a logical or condition (Production 55). The resulting expression’s true list is a merge of the true lists of both of the binary expressions. The false list of the resulting expression is the false list of the second binary expression. In a similar fashion the AND_OP (&&) is used to combine the Boolean expressions (Production 58). The resulting expression’s true list is the true list of the second Boolean expression. The resulting expression’s false list is a merge of the false lists of both the Boolean expressions.

55	or_expr	:	or_expr OR_OP M and_expr { backpatch(or_expr$_1$.falselist, M.lbl)); quadlist_merge(or_expr.truelist,or_expr$_1$.truelist, and_expr.truelist); or_expr.falselist = and_expr.falselist ; }

58	and_expr	|	and_expr AND_OP M binary_expr {
			`backpatch(and_expr`$_1$`.truelist, M.lbl);`
			`and_expr.truelist = binary_expr.truelist ;`
			`quadlist_merge(and_expr.falselist,and_expr`$_1$`.falselist,`
			`binary_expr.falselist);`
			}

The non-terminal 'stmt' takes the form of a simple if statement (Production 48), or if-else statement (Production 49). The stmt can take the form of a compound statement (Production 50), which is essentially a group of statements within the braces. A stmt non-terminal has an attribute *nextlist,* which is a list of TAC statements generated by 'stmt' containing the conditional and unconditional jumps to the quadruple following the statement. This is an important attribute using which the intermediate code can be backpatched for jumps during if, if-else and while constructs.

The conditional statement for 'if' construct (Production 48) makes use of a marker M. The marker helps determine the first statement to be executed, if the Boolean expression E is true. In case the value of Boolean expression is false, then the target label for the jump is the statement following the 'if' statement. Hence, it adds the test_expr.falselist into the statement's nextlist. This is given by quadlist_merge(stmt.nextlist, test_expr.falselist,stmt1.nextlist) in the semantic action.

The if-else statement (Production 49) uses two of the markers M to know the first executable statement when the Boolean expression is true and false respectively. For representing the attributes the marker M following the 'test-expr' is treated as M_1 and the M following the 'ELSE' keyword is treated as M_2. They are backpatched by using the function backpatch(test_expr.truelist, M_1.quad), backpatch(test_expr.falselist, M_2.quad). After generating TAC instructions for all the statements within the 'if' condition, a goto statement is generated to jump to the following statement after the 'if–else' construct. The marker N helps achieve this. The label of the goto statement is added to 'stmt.nextlist' to reflect the conditional/unconditional goto TAC statements that jump to the next statement.

48	stmt	:	IF '(' test_expr ')' M stmt {
			`backpatch(test_expr.truelist, M.lbl));`
			`quadlist_merge(stmt.nextlist, test_expr.falselist,stmt`$_1$`.nextlist);`
			}

49		|	IF '(' test_expr ')' M stmt ELSE N M stmt {
			`backpatch(test_expr.truelist,M`$_1$`.lbl);`
			`backpatch(test_expr.falselist,M`$_2$`.lbl);`
			
			`quadlist_merge(tmp, N.nextlist,stmt`$_2$`.nextlist);`
			`quadlist_merge(stmt.nextlist,stmt`$_1$`.nextlist,tmp);`
			}

50	stmt	:	'{' stmt_list '}' {
			`stmt.nextlist = stmt_list.nextlist;`
			}

66	N	:	ε {
			`N.nextlist = makelist(next_quad) ;`
			`emit (GOTO,NULL,NULL,NULL);`
			}

A marker M is inserted in several productions of the grammar above to get the label of the next quadruple to be generated. The marker M facilitates in getting the label of next TAC statement to be generated by means of Production 65. The label of next TAC statement to be generated is stored in M.lbl. In Production 48 (also in 49), the attribute M.lbl is used to backpatch the TAC statements that have an unfilled goto's with the value in M.lbl.

65	M	:	∈ { lbl = newlbl(); emit(LBL,NULL,NULL,lbl); M.lbl = lbl ; }

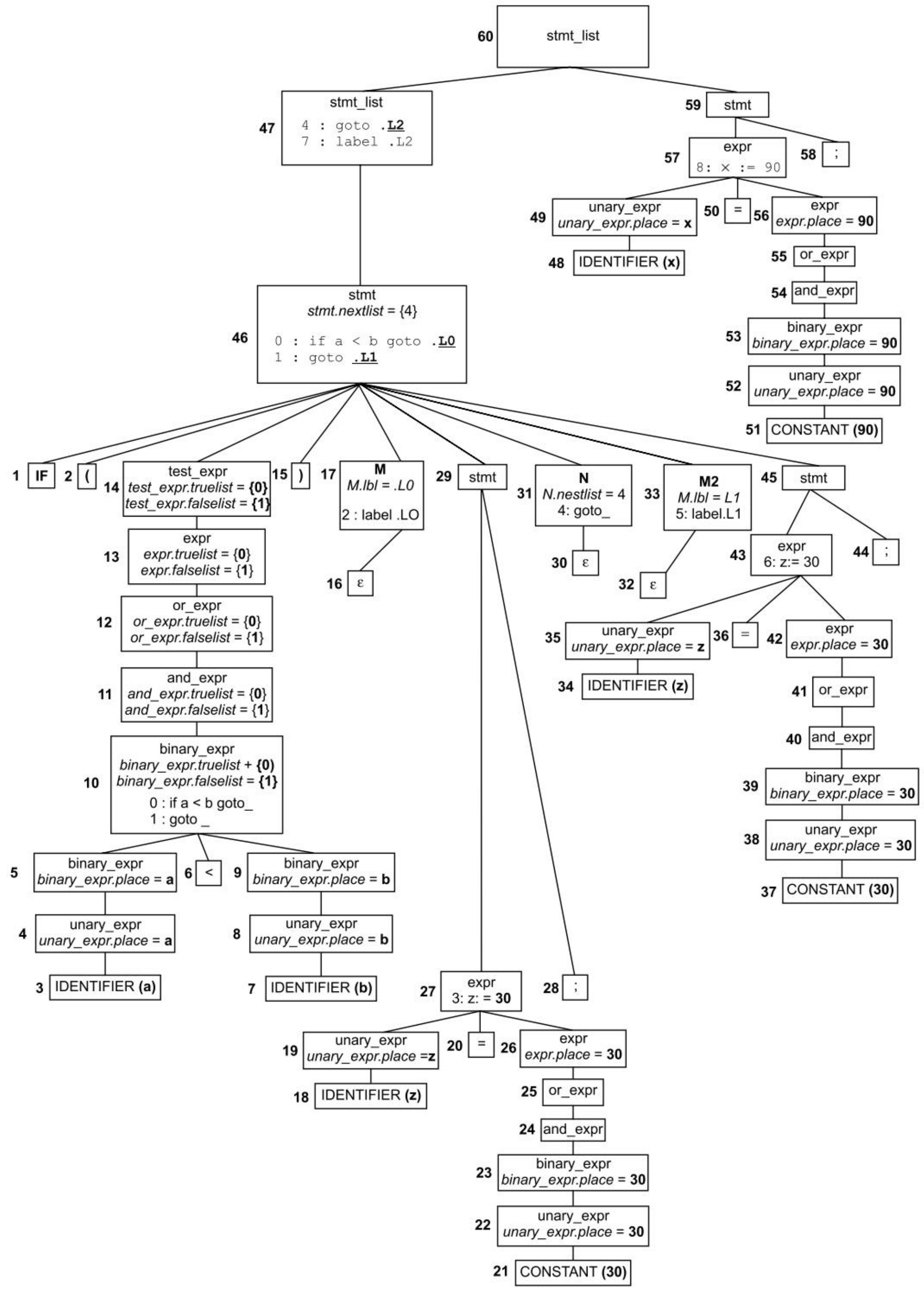
The statements that have the next_list attribute as non-empty indicate that there are some quads already generated, that have the target label as the next statement. For example, Production 48 and 49 generate quads whose target label is the next following statement. The attribute next_list of the 'stmt' contains the list of such quads whose target label is the next following statement. Production 51 and 52 generate a label for the next statement and do backpatching for those quads.

51	stmt_list	:	stmt_list stmt { if(stmt.nextlist is not empty){ lbl = newlbl(); emit(LBL,NULL,NULL,sptr); backpatch(stmt_list$_1$.nextlist,lbl); } }
52		|	stmt { if(stmt.nextlist is not empty){ lbl = newlbl(); emit(LBL,NULL,NULL,sptr); backpatch(stmt.nextlist,lbl); } }

Table 5.24 shows an if-else statement and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.19 shows the attributes of the different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation.

Table 5.24 *Input source and its translated code*

C code snippet	Translated TAC
`if ( a < b )`	`0: if a < b goto .L0`
`    z = 30 ;`	`1: goto .L1`
`else`	`2: label .L0`
`    z = 40 ;`	`3: z := 30`
	`4: goto .L2`
`x=90;`	`5: label .L1`
	`6: z := 40`
	`7: label .L2`
	`8: x := 90`



60 stmt_list
47 stmt_list
4 : goto .L2
7 : label .L2
59 stmt
57 expr
8: x := 90
58 ;
49 unary_expr unary_expr.place = x
50 =
56 expr expr.place = 90
48 IDENTIFIER (x)
55 or_expr
54 and_expr
53 binary_expr binary_expr.place = 90
52 unary_expr unary_expr.place = 90
51 CONSTANT (90)
46 stmt stmt.nextlist = {4}
0 : if a < b goto .L0
1 : goto .L1
1 IF
2 (
14 test_expr test_expr.truelist = {0} test_expr.falselist = {1}
15)
17 M M.lbl = .L0 2 : label .LO
16 ε
29 stmt
31 N N.nestlist = 4 4: goto_
30 ε
33 M2 M.lbl = L1 5: label.L1
32 ε
45 stmt
43 expr 6: z:= 30
44 ;
13 expr expr.truelist = {0} expr.falselist = {1}
12 or_expr or_expr.truelist = {0} or_expr.falselist = {1}
11 and_expr and_expr.truelist = {0} and_expr.falselist = {1}
10 binary_expr binary_expr.truelist + {0} binary_expr.falselist = {1} 0 : if a < b goto_ 1 : goto _
5 binary_expr binary_expr.place = a
6 <
9 binary_expr binary_expr.place = b
4 unary_expr unary_expr.place = a
8 unary_expr unary_expr.place = b
3 IDENTIFIER (a)
7 IDENTIFIER (b)
35 unary_expr unary_expr.place = z
36 =
42 expr expr.place = 30
34 IDENTIFIER (z)
41 or_expr
40 and_expr
39 binary_expr binary_expr.place = 30
38 unary_expr unary_expr.place = 30
37 CONSTANT (30)
27 expr 3: z: = 30
28 ;
19 unary_expr unary_expr.place =z
20 =
26 expr expr.place = 30
18 IDENTIFIER (z)
25 or_expr
24 and_expr
23 binary_expr binary_expr.place = 30
22 unary_expr unary_expr.place = 30
21 CONSTANT (30)

Fig. 5.19 *Translation of if-else construct*

5.2.10 Example 5—Translation of if, if-else statements

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving if and if-else constructs using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with some statements involving if and if-else constructs. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test5.c
     1  int x,z;
     2
     3  int
     4  func (int a, int b)
     5  {
     6    if (a < b) {
     7        z = 30;
     8    } else {
     9        z = 40;
    10    }
    11    x=90;
    12
    13  }


# Generating IC
$ ./icgen test5.c
 (0) proc_begin func
 (1) if a < b goto .L0
 (2) goto .L1
 (3) label .L0
 (4) z := 30
 (5) goto .L2
 (6) label .L1
 (7) z := 40
 (8) label .L2
 (9) x := 90
(10) label .L3
(11) proc_end func


# Input file
$ cat -n test5a.c
    1 int x,z;
    2
    3 int
    4 func (int a, int b, int c, int d)
    5 {
    6   if ((a < b) || ( c < d )) {
    7       z = 30;
    8   } else {
    9       z = 40;
   10   }
   11   x=90;
   12
   13 }

# Generating IC
$ ./icgen test5a.c
 (0) proc_begin func
 (1) if a < b goto .L1
 (2) goto .L0
 (3) label .L0
 (4) if c < d goto .L1
 (5) goto .L2
 (6) label .L1
 (7) z := 30
 (8) goto .L3
 (9) label .L2
(10) z := 40
(11) label .L3
(12) x := 90
(13) label .L4
(14) proc_end func

# Input source file
$ cat -n test5b.c
    1 int x,z;
    2
    3 int
    4 func (int a, int b, int c, int d)
    5 {
    6   if ((a < b) && ( c < d )) {
    7       z = 30;
    8   } else {
    9       z = 40;
   10   }
   11   x=90;
   12
   13 }

# Generating IC
```

```
$ ./icgen test5b.c
 (0) proc_begin func
 (1) if a < b goto .L0
 (2) goto .L2
 (3) label .L0
 (4) if c < d goto .L1
 (5) goto .L2
 (6) label .L1
 (7) z := 30
 (8) goto .L3
 (9) label .L2
(10) z := 40
(11) label .L3
(12) x := 90
(13) label .L4
(14) proc_end func
```

5.2.11 While Statement

In this section, we learn about the translation of another flow of control statement—the while statement. The while statement is used in higher level languages to repeatedly execute a section of code as long as the entry condition is satisfied. The common way to break out of the while statement loop is when the entry condition is no longer satisfied, or when there is a explicit 'break' statement executed by the program.

Consider the C code snippet containing a while statement in Table 5.25 for understanding the important aspects of intermediate code generation for a while statement. The variables used in the code snippet are all assumed to be integers. The entry condition to the while loop is given by the test expression ($i > 0$). The statements val = val *i and $i = i - 1$; are executed repeatedly as long as the entry condition is satisfied.

Table 5.25 *C code snippet using while statement*

```
..
..
while(i>0)
{
    val = val * i;
    i = i-1;
}
..
..
```

Table 5.26 shows the intermediate code for the code snippet in Table 5.25. We can see from Table 5.26 that the generated intermediate code for the while loop is very similar to the 'if' statement with an additional jump to the entry condition.

Table 5.26 *TAC statements for 'while' statement*

```
0 : label .L0
1 : if i > 0 goto .L1
2 : goto .L2
3 : label .L1
4 : _t0 := val * i
5 : val := _t0
6 : _t1 := i - 1
7 : i := _t1
8 : goto .L0
9 : label .L2
```

The translation of a 'while' statement is similar to the if-else construct that we saw in the last section. The backpatch technique is used again for overcoming the issue of unknown labels at the time of translation.

We now look at the productions and the corresponding semantic actions relevant for the translation of a while statement.

The while statement (see Production 67 below) uses a marker R to know the label of the beginning statement of the 'while' loop. The beginning statement of a while loop is the test expression statement which is the entry condition for the while loop. It uses another marker M to determine the label of the first executable statement to jump, if the value of test expression is true. Both of these labels are backpatched on the completion of processing of the 'while' statement. The statement to jump when the Boolean expression is false is the next executable statement after the while loop. Hence, the test_expr.falselist is added into the statement's nextlist along with break labels in the processing of Production 67. At the end of TAC statements for while loop, a goto statement that jumps back to the start of while loop to check the entry condition (marked by R) is generated.

67	stmt	:	WHILE R '(' test_expr ')' M stmt {
			`while (top of Break-Label stack is not marker)`
			`{`
			`    pop the Label.`
			`    add the label to brk_list`
			`}`
			`backpatch(stmt`$_1$`.nextlist,R.lbl));`
			`backpatch(test_expr.truelist, M.lbl);`
			`quadlist_merge(stmt.nextlist, test_expr.falselist,brk_list) ;`
			`emit (GOTO,NULL,NULL,R.quad);`
			}

69	R	:	∈ {
			`b = new brk_lbl(-1); /* marker */`
			`brk_lbl_stk.push(b);`
			`lbl = newlbl();`
			`emit(LBL,NULL,NULL,lbl);`
			`R.lbl = lbl;`
			}

The 'break' statement processing involves generating an unfilled 'goto' statement (Production 68). The label is pushed on to a ***break-label stack*** to signify that it needs to be backpatched later. The labels of the break statements are consolidated and added to the 'stmt.nextlist' during the processing of the while statement in Production 67. The break label stack uses a marker to differentiate between the nested while statements and their corresponding break statements as seen in the processing of Production 69.

68	stmt	:	BREAK ';' {
			`b = new brk_lbl(next_quad);`
			`brk_lbl_stk.push(b);`
			`emit (GOTO,NULL,NULL,NULL);`
			}

Table 5.27 shows an input source containing a 'while' statement and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.20 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation.

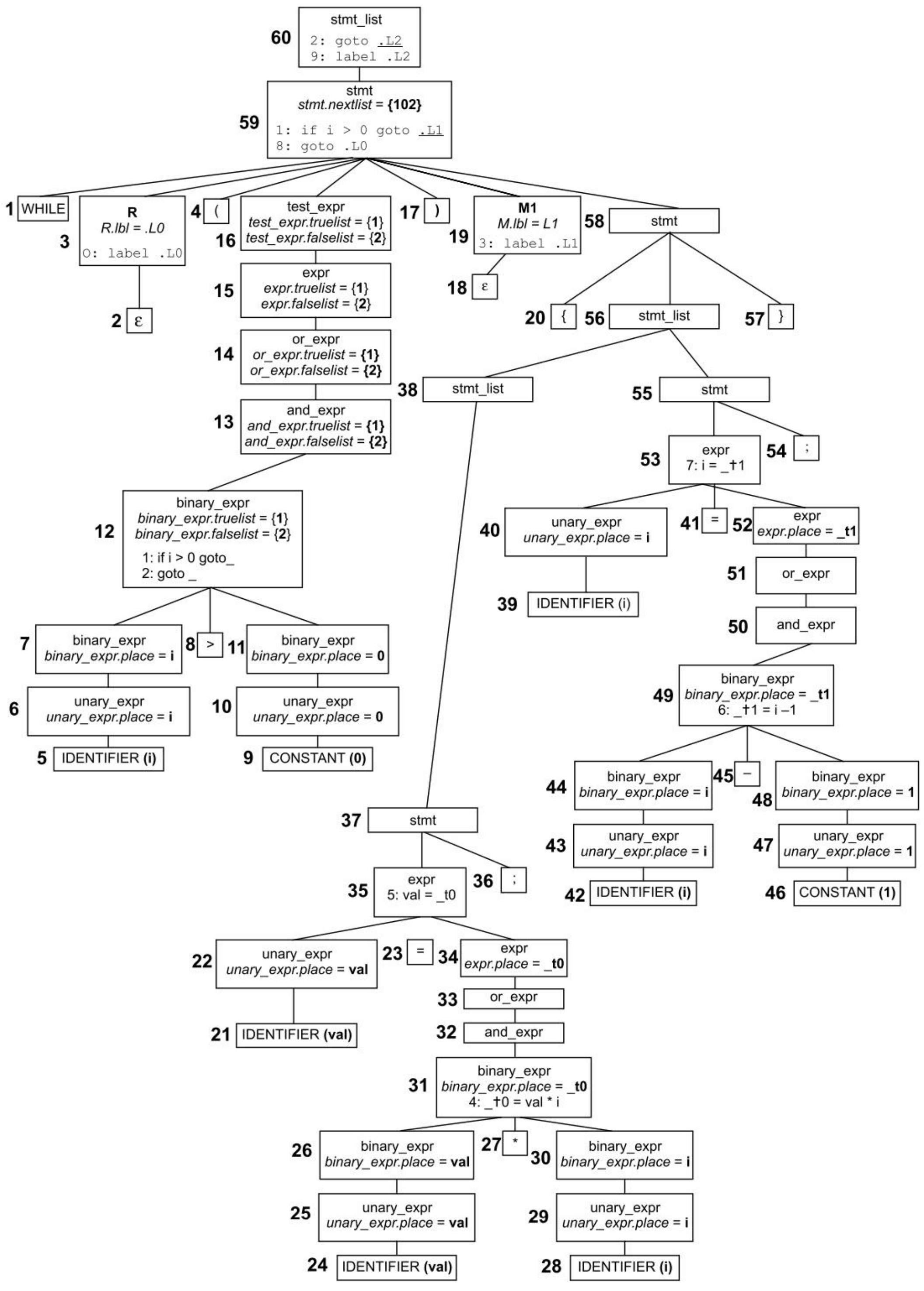


Fig. 5.20 *Translation of 'while' construct*

Table 5.27 *Input source and its translated code*

C code snippet	Translated TAC
`while ( i > 0 )`	`0 : label .L0`
`{`	`1 : if i > 0 goto .L1`
`    val = val * i;`	`2 : goto .L2`
`    i = i -1 ;`	`3 : label .L1`
`}`	`4 : _t0 := val * i`
	`5 : val := _t0`
	`6 : _t1 := i - 1`
	`7 : i := _t1`
	`8 : goto .L0`
	`9 : label .L2`

5.2.12 Example 6—Translation of While Statements

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for while statement using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with some switch-case statements. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test6.c
     1 int val;
     2
     3 int func(int num)
     4 {
     5   int i,val;
     6
     7   val=1;
     8   i=num;
     9
    10   while(i>0)
    11   {
    12        val = val * i;
    13          i = i-1;
    14    }
    15
    16 }

# Generating IC
$ ./icgen test6.c
 (0) proc_begin func
 (1) val := 1
 (2) i := num
 (3) label .L0
 (4) if i > 0 goto .L1
 (5) goto .L2
 (6) label .L1
 (7) _t0 := val * i
 (8) val := _t0
 (9) _t1 := i - 1
(10) i := _t1
(11) goto .L0
(12) label .L2
(13) label .L3
(14) proc_end func

# Input file
$ cat -n test6a.c
     1 int val;
     2
     3 int func(int num)
     4 {
     5         int i,val;
     6
     7         val=1;
     8         i=num;
     9
    10         while(i>0)
    11         {
    12                 val = val * i;
    13                 i = i-1;
    14
    15                 if(val > 10000 ){
    16                         break;
    17                 }
    18         }
    19
    20 }

# Generating IC
$ ./icgen test6a.c
 (0) proc_begin func
 (1) val := 1
 (2) i := num
 (3) label .L0
 (4) if i > 0 goto .L1
 (5) goto .L4
 (6) label .L1
 (7) _t0 := val * i
 (8) val := _t0
 (9) _t1 := i - 1
(10) i := _t1
(11) if val > 10000 goto .L2
(12) goto .L3
(13) label .L2
(14) goto .L4
(15) label .L3
(16) goto .L0
(17) label .L4
(18) label .L5
(19) proc_end func
```

5.2.13 Switch-case Statement

The switch-case statement is another common programming construct found in most of the programming languages. It is used in scenarios where there is a need to execute a set of statements depending on the value of an expression. The switch case statement is supported in C language.

Consider the C code snippet containing a switch-case in Table 5.28 for understanding the important aspects of intermediate code generation for a switch-case statement. The variables used in the code snippet are all assumed to be integers.

Table 5.28 *Switch-case code snippet in C language*

```
..
..

switch(x+y)
{
        case 3 :
                a=b;
                c=d;
                break;

        case 5 :
                e=f;
                g=h;
        case 7 :
                i=j;
                k=l;
                break;

        default :
                a=b;
                c=d;
                break;
}
p=q
..
..
```

We shall spend some time understanding the components of the switch case statement before we try to attempt a translation of the switch-case to TAC statements.

The switch statement consists of three main parts:

- The selector expression based on whose value the control flows to a specific case statement. In the code snippet shown in Table 5.28, the selector expression is $(x + y)$.
- The set of statements to be executed when the selector expression matches a particular case expression. In the code snippet shown in Table 5.28, the set of statements $a = b$; $c = d$; break; get executed when the selector expression, i.e. $(x + y)$ is equal to the case expression 3. The break statement allows the control to jump out of the switch statement to the next executable statement, which is $p = q$;
- A 'default' case, which allows for executing a set of statements in the event of the selector expression not matching any of the case expressions. In the above example, the statements in the 'default' get executed if the selector expression $(x + y)$ is not equal to any of the case expressions 3 or 5 or 7. The default case is optional in C language.

Table 5.29 shows the intermediate code for the switch-case statement shown in Table 5.28. The index in the quads array is also shown alongside the instruction for reference.

Table 5.29 *TAC statements for the switch-case statement*

```
 0:  t1 := x + y
 1:  goto .L4
 2:  label .L0
 3:  a := b
 4:  c := d
 5:  goto .L5
 6:  label .L1
 7:  e := f
 8:  g := h
 9:  label .L2
10:  i := j
11:  k := l
12:  goto .L5
13:  label .L3
14:  a := b
15:  c := d
16:  goto .L5
17:  goto .L5
18:  label .L4
19:  if t1 == 3 goto .L0
20:  if t1 == 5 goto .L1
21:  if t1 == 7 goto .L2
22:  goto .L3
23:  label .L5
24:  p := q
```

The translation of the switch case statement would broadly be divided into three parts as follows:

1. Generation of TAC statements to evaluate the selector expression. In Table 5.29, the quad at the index 0 evaluates the selector expression and stores it in the temporary *_t0* for the upcoming comparisons.
2. Generation of TAC statements for each one of the cases in the switch statement. Each case is associated with a unique label as an entry point. In Table 5.29, the TAC statements 2 through 5 are generated for the case '3' in the input source. The entry point for the case 3 is the label .L0. Similarly, the statements 9 through 12 are associated with the case '7' in the input source with the entry point being the label .L2.

3. Generation of a set of 'if' TAC statements to compare the selector expression with each one of the case expressions. These conditional statements jump to the respective entry point label identified in (2) above on success. In Table 5.29, the TAC statements 19 through 22 compare the selector expression in _t0 with each of the case expressions 3, 5 and 7. The 'if' statement in 19 jumps to the entry point label .L0 associated with the case expression 3 on success. The 'if' statement in 20 jumps to the entry point label .L1 associated with case expression 5 on success. The quad in 22 is for the default label of the case statement.

We now look at the productions and the corresponding semantic actions relevant for the translation of a switch-case statement.

A value-label (VL) stack is used to maintain the mapping between the value of the selector expression and the label to be jumped for that particular value of selector expression. Each time the parser encounters a case statement, the value of selector expression and the label for the first statement to be executed corresponding to that value are stored on the value-label stack. For example, when case 5 is encountered in the input source shown in Table 5.28, the value 5 and the label of the next statement, i.e. *e* = *f*, given by M.lbl are stored on the VL stack. This is manifested in the semantic computation associated with Production 71. The default case (Production 72) is also handled similarly except the fact that in the value-label pair, the value does not hold significance; an unconditional goto is generated to the label of the 'default' case.

71	case_stmt	:	CASE CONSTANT ':' M stmt_list {
			 `push the pair M.lbl,CONSTANT.value on VL stack`
			 }

72		|	DEFAULT ':' M stmt_list {
			 `push the pair M.lbl,default on VL stack`
			 }

73	case_stmt_list	:	case_stmt_list case_stmt
74		|	case_stmt

After the parser identifies the 'switch' keyword, a marker is pushed on the value-label stack. This serves as a demarcation on the value-label stack for the elements corresponding to this particular switch-case statement. This is manifested in the semantic processing of Production 75. Having a marker on value-label stack enables us to handle nested switch case statements. After processing the complete switch statement including all the cases, a series of 'if' TAC statements are generated, which check the value of selector expression with a jump to appropriate label on success. This is done using the VL stack that was populated during the handling of case statements. The semantic processing associated with the Production 70 is indicative of this.

The processing for 'break' statement seen in Production 68 earlier in Section 5.3.11 remains unchanged. The labels of the break statements are consolidated and added to the 'stmt.nextlist' during the processing of the switch statement in Production 70.

In the TAC statements shown in Table 5.29, the lines associated with indexes 17 through 21 are generated from semantic processing of Production 70.

70	stmt	:	SWITCH Q expr N '{' case_stmt_list '}' {
			 `list1 = makelist(nextquad);`
			 `emit (GOTO,NULL,NULL,NULL);`

			backpatch(N.nextlist,nextquad);
			
			lbl = newlbl();
			emit(LBL,NULL,NULL,lbl);
			
			while (top of Value-Label stack is not marker)
			{
			    pop the Value-Label pair VL.
			    if (VL is default case){
			        ldef = L
			        emit('goto', ldef);
			    } else
			        emit(EQ,expr.place,value,label)
			    }
			}
			while (top of Break-Label stack is not marker)
			{
			    pop the Label.
			    add the label to list2
			}
			quadlist_merge(stmt.nextlist,list1,list2)
			}

75	Q	:	∈ {
			    push the marker on VL stack
			    push the marker on the Break Label Stack
			}

The backpatching technique is used to fill in the label for the next executable statement for the list of statements given by attribute stmt.nextlist, filled in Production 70. The backpatching happens in the processing of Production 50/51 seen earlier during the discussion on if-constructs in Section 5.3.9.

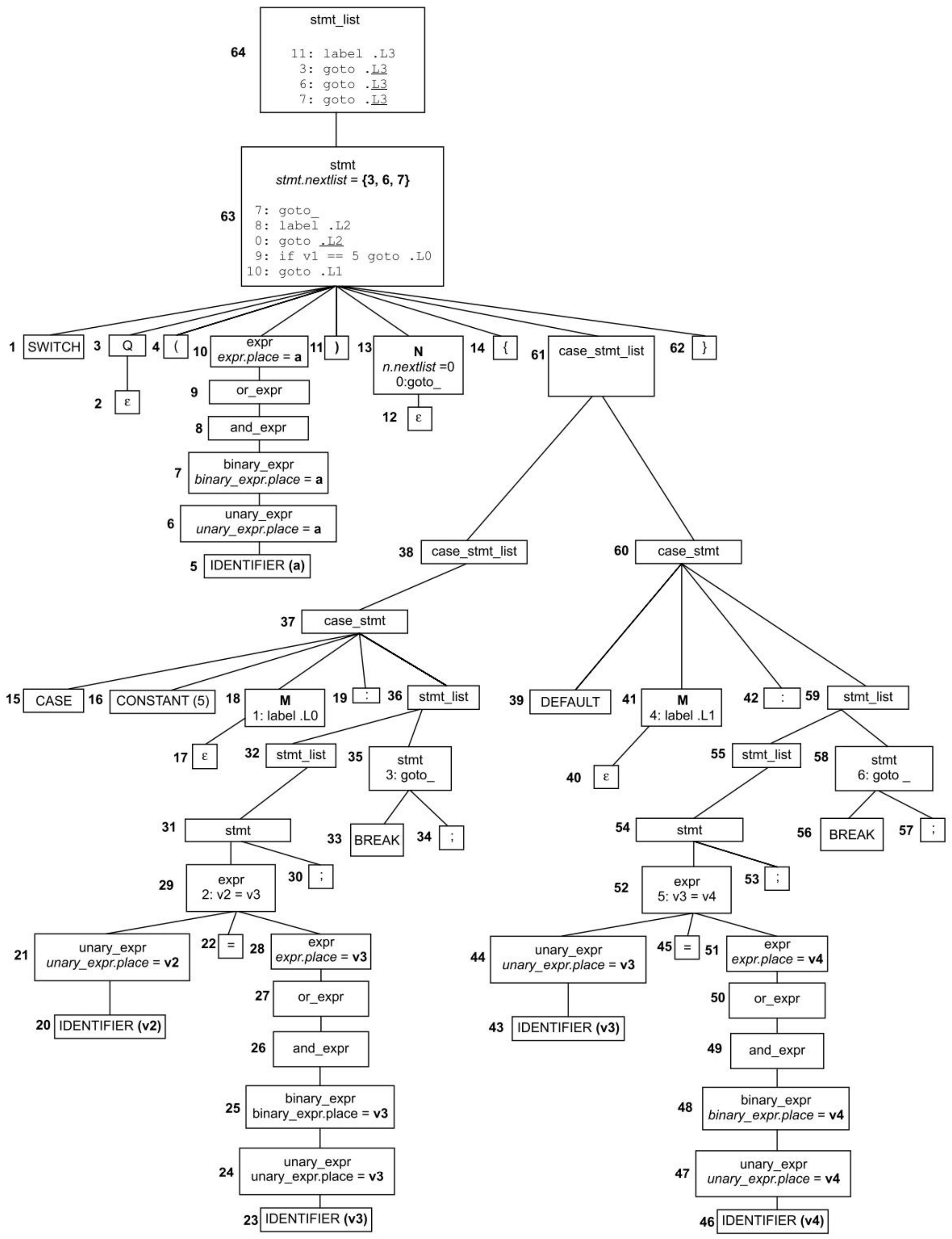
In the TAC statements shown in Table 5.29, the lines associated with indexes 5, 12, 16 and 17 are all backpatched by the processing of Production 50 or 51. The 'stmt.nextlist' for the switch-case statement at the end of processing Production 70 has the four elements 5,12,16 and 17 corresponding to Table 5.29.

Table 5.30 shows a C code snippet containing a 'switch-case' construct and its corresponding translated code using the productions and semantic actions discussed above. Figure 5.21 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation.

Table 5.30 *Input source and its translated code*

Input source	Translated TAC

switch(v1)
{
    case 5   : v2=v3;
               break;
    default : v3=v4;
               break;
}
	0: goto .L2
	1: label .L0
	2: v2 := v3
	3: goto .L3
	4: label .L1
	5: v3 := v4
	6: goto .L3
	7: goto .L3
	8: label .L2
	9: if v1 == 5 goto .L0
	10: goto .L1
	11: label .L3



```
11: label .L3
 3: goto .L3
 6: goto .L3
 7: goto .L3
```

```
 7: goto_
 8: label .L2
 0: goto .L2
 9: if v1 == 5 goto .L0
10: goto .L1
```

Fig. 5.21 *Translation of switch-case construct*

5.2.14 Example 7—Translation of Switch-case Statements

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for switch-case statements using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with some switch-case statements. The output of 'icgen' is the intermediate code in TAC format generated from processing the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l


# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc


# Building icgen Binary
$ g++   -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen


# Input file
$ cat -n test7.c
    1  int z;
    2
    3  int
    4  func (int sel_exp, int a, int b)
    5  {
    6
    7       switch (sel_exp)
    8       {
    9           case 5:
   10                  z = a + b;
   11                  break;
   12           default:
   13                  z = a - b;
   14                  break;
   15       }
   16       z = z * b;
   17  }

# Generating IC
$ ./icgen test7.c
 (0) proc_begin func
 (1) goto .L2
 (2) label .L0
 (3) _t0 := a + b
 (4) z := _t0
 (5) goto .L3
 (6) label .L1
 (7) _t1 := a - b
 (8) z := _t1
 (9) goto .L3
(10) goto .L3
(11) label .L2
(12) if sel_exp == 5 goto .L0
(13) goto .L1
(14) label .L3
(15) _t2 := z * b
(16) z := _t2
(17) label .L4
(18) proc_end func

# Input file
$ cat -n test7a.c
    1  int z;
    2
    3  int
    4  func (int sel_exp, int a, int b)
    5  {
    6
    7    switch (sel_exp)
    8    {
    9      case 5:
   10        z = a + b;
   11        break;
   12      case 6:
   13        z = a + 2*b;
   14        break;
   15      case 7:
   16        z = a + 3*b;
   17        break;
   18      default:
   19        z = a - b;
   20        break;
   21    }
   22    z = z * b;
   23  }

# Generating IC
$ ./icgen test7a.c
 (0) proc_begin func
 (1) goto .L4
 (2) label .L0
 (3) _t0 := a + b
 (4) z := _t0
 (5) goto .L5
 (6) label .L1
 (7) _t1 := 2 * b
 (8) _t2 := a + _t1
 (9) z := _t2
(10) goto .L5
(11) label .L2
```

```
(12) _t3 := 3 * b
(13) _t4 := a + _t3
(14) z := _t4
(15) goto .L5
(16) label .L3
(17) _t5 := a - b
(18) z := _t5
(19) goto .L5
(20) goto .L5
(21) label .L4
(22) if sel_exp == 5 goto .L0
(23) if sel_exp == 6 goto .L1
(24) if sel_exp == 7 goto .L2
(25) goto .L3
(26) label .L5
(27) _t6 := z * b
(28) z := _t6
(29) label .L6
(30) proc_end func
```

5.2.15 Procedure Calls

In this section, we study about the translation of procedure or function calls. The functions are used in higher level languages to break down a large program into smaller modular components. Strictly speaking, a function is a procedure with a return value, but for the discussion here, we use the terms function and procedure interchangeably.

Consider a small C code snippet that calls a function ‘my_func’ as shown in Listing 5.2 to understand the important aspects of intermediate code generation for a function call. For the sake of simplicity, it is assumed that all the variables used in the code snippet are integers that require 4 bytes of memory each.

```
..
..
v3=my_func((v1+6),v2);
v4=v3+5;
..
```

Listing 5.2 *C Code snippet calling a function*

There are several things that happen, when a function is called or invoked in the input source as shown in Listing 5.2.

- The arguments to the called function are evaluated. For example, in the code snippet of Listing 5.2, the arguments ‘*v*1 + *6*’ and *v2* are evaluated. The argument list consists of expressions, which would include, arguments by reference, pointers, arithmetic expressions, and so on. The attribute expr.place would typically hold the evaluation of an argument expression.
- Each of the arguments is put into a place in memory from where they can be accessed by the called function. In the code snippet of Listing 5.2, the evaluated arguments, namely (*v*1 + 6) and *v*2 are put into a pre-determined place in memory for the function ‘my_func’ to access them. The ‘***param***’ TAC statement does the job of putting the argument into a pre-determined place in memory to be accessed by the calee. A ‘param’ statement is emitted for every argument that is used by the procedure.
- The control is transferred to the first statement of the procedure called. But, just before that, the ‘return’ address, where the control should return after executing the procedure needs to be put into

a place in memory, so that there is a smooth transfer of control back after the function is executed. The *'call'* TAC statement does the job of storing the return address in a pre-determined place and then transferring the control to the first statement of the procedure called. The cumulative size of the arguments in bytes for each parameter generated by 'param' statements is the other argument for 'call' statement. It is used during the target code generation.

From a TAC statements point of view, a function call invocation in C language is translated into the following: (a) TAC statements for evaluation of each of the arguments; (b) A series of '***param***' statements, one for every argument that is used during the invocation of a function; (c) A *'**call**'* statement, which involves saving the return address and transferring the control to the first statement in the function.

This sequence of TAC instructions generated for a function invocation is termed as the '***calling sequence***'. Table 5.31 shows the section of translated code corresponding to the code snippet in Listing 5.2. It illustrates the calling sequence that we just discussed.

Table 5.31 *TAC statements illustrating the calling sequence*

Translated TAC
`_t0 := v1 + 6   /* Evaluating Argument */`
`param v2`
`param _t0`
`call my_func 8  /* 8 is the cumulative size of arguments`
`                   in bytes generated by param statements (v2 and _t0 )*/`

After the function is executed and the control returns back to the caller, the returned value is fetched from the pre-determined place into a local variable. This is achieved by a *'**retrieve**'* TAC statement. The argument to the retrieve TAC statement is the variable, where the returned value needs to be stored.

The 'retrieve' TAC statement can be envisaged as the ***returning sequence*** similar to the calling sequence we studied earlier. Table 5.32 shows the returning sequence incorporated along with the calling sequence for the translation of input source of Listing 5.2.

Table 5.32 *TAC statements illustrating both the calling and the returning sequence*

```
/* Calling Sequence */
_t0 := v1 + 6
param v2
param _t0
call add_func

/* Returning Sequence */
retrieve _t1

v3 := _t1
v4 := v3+5
```

In similar lines, when a function gets called, there is sequence of actions that happen in the ***called*** function.

- The called function saves the registers and accommodates storage for local variables. This is achieved by the '***proc_begin***' TAC statement, which has the name of the function as an argument.
- The called function can have a 'return' statement of the form *return(expression)* in the input source. This is translated to (a) placing the return value into a place, where the caller wants it, and (b) jumping to the end of the function. The placing of return value into a place agreed by caller is achieved by the

'return' TAC statement. The argument to the return TAC statement specifies the value that is being returned. The jump to the end of the function is performed by a '*goto*' TAC statement. If the input C source contains a return statement without the expression, only a goto TAC statement is generated, which accomplishes (b) above.

- The end of a function is marked by *'proc_end'* TAC statement. The proc_end TAC statement would be responsible for restoring the registers, release the local space and transfer the control back to the caller. A label signifying the end of the function (say end_lbl) is also generated before the proc_end TAC statement. The end_lbl is used as the target of all the goto statements generated with 'return' statements mentioned above.

Table 5.33 shows a simple function 'add_func' and its corresponding translated intermediate code illustrating the concepts discussed above. The 'add_func' adds the two parameters and returns the result.

Table 5.33 *A function and its translation*

C Code Snippet	Translated TAC
`int add_func(int a, int b)`	`0: proc_begin add_func`
`{`	`1: _t0 := a + b`
`     int c;`	`2: c := _t0`
	`3: return c`
`     c = a + b;`	`4: goto .L0`
`     return(c);`	`5: label .L0`
`}`	`6: proc_end add_func`

Having understood the TAC statements relevant to procedure calls, let us now look at the productions and the semantic actions relevant for the translation of procedure calls.

A queue called 'args_queue' is a placeholder for all the actual parameters that are used in invoking the function. The args_queue is initialised, when the first argument is spotted in Production 79 (see below). Every argument succeeding it is appended into the args_queue in Production 78. When the function call is translated in Production 77, a 'param' statement is generated for each of the argument stored in the args_queue.

A function invocation (Productions 76, 77) results in a 'call' TAC statement. The 'call' TAC statement is preceded by the series of param statements in case there are arguments to the function as seen in the semantic action for Production 77. The cumulative size of parameters in bytes is an argument for the CALL TAC statement. For functions that have a return non-void return type, a 'retrieve' TAC statement is generated. This is shown in the code segments of both Productions 76 and 77.

76	unary_expr	:	unary_expr '(' ')' {
			`param_bytes = make_lit_tbl_entry(0);`
			`emit(CALL, u->place,param_bytes,NULL);`
			`if(unary_expr`$_1$`.place.type == FUNCTION ){`
			`    f = unary_expr`$_1$`.place.type;`
			`    if(f.ret_type!= VOID_TYPE ){`
			`        t = newtemp1(f.ret_type);`
			`        emit(RETRIEVE,NULL,NULL,t);`
			`        unary_expr.place = t ;`
			`    }`
			`}`
			}

77		|	unary_expr '(' args_list ')' {
			    for (each element s in args_list.args_queue){
			        emit(PARAM,NULL,NULL,s);
			        sz = sz + sizeof(s)
			    }
			    param_bytes = make_lit_tbl_entry(sz);
			
			    emit(CALL,unary_expr$_1$.place,param_
			    bytes,NULL,NULL);
			    if(unary_expr$_1$.place.type == FUNCTION){
			        f = (function *)unary_expr$_1$.place.type;
			        if(f.ret_type!= VOID_TYPE){
			            t = newtemp1(f.ret_type);
			            emit(RETRIEVE,NULL,NULL,t);
			            unary_expr.place = t ;
			        }
			    }
			}

78	args_list	:	args_list ',' expr {
			    Append expr.place to args_list.args_queue;
			}

79		|	expr {
			    Append expr.place into args_list.args_queue
			}

A simple return statement without parameter is dealt with in Production 80, by generating a goto TAC statement. The target label for the goto TAC statement is the end of the function, which is backpatched in Production 5 by means of using a list 'return_list'.

The return statement with a parameter is dealt with in Production 81, where return TAC statement is generated with expr.place (that holds the return value) as the argument. The return TAC statement is followed up by generating a goto TAC statement. The target label for the goto TAC statement is the end of the function, which is backpatched in Production 5 by means of using a list 'return_list'.

80	stmt	:	RET ';' {
			    Add next_quad to return_list;
			    emit(GOTO,NULL,NULL,NULL);
			}

81		|	RET expr ';' {
			    emit(RETURN,NULL,NULL,expr.place);
			    Add next_quad to return_list;
			    emit(GOTO,NULL,NULL,NULL);
			}

The translated code for a function definition (Production 5 and 82) contains the generation of a ***'proc_begin'*** TAC statement signifying the start of a function. A label TAC statement followed by the '***proc_end***' TAC statement is generated in Production 5 signifying the end of the definition of a function.

5	function_definition	:	type_spec func_decl P function_body {
			`curr_sym_tab_ptr = tbl_stk.top();`
			`Pop tbl_stk ;`
			`emit(LBL,NULL,NULL,end_label);`
			`backpatch(return_list,end_label)`
			`emit(PROC_END,NULL,NULL,func_name);`
			}

82	P	:	∈ {
			`Push curr_sym_tab_ptr on tbl_stk`
			`curr_sym_tab_ptr = new symbol table ;`
			`curr_sym_tab_ptr->previous = tbl_stk.top() ;`
			`emit (PROC_BEGIN,NULL,NULL,func_name);`
			}

Table 5.34 shows a a procedure definition and its corresponding translated code generated using the productions and semantic actions discussed above. Figure 5.22 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation. The func_decl and the declaration_list non-terminals are not shown in expanded form in the figure, they don't emit any TAC instructions.

Table 5.34 *Input source and its translated code*

Input Source	Translated TAC

`int add_func(int a,int b)`
`{`
`        int c;`
 	
`        c = a + b;`
 	
`        return c;`
`}`
	`0: proc_begin add_func`
	`1: _t0 := a + b`
	`2: c := _t0`
	`3: return c`
	`4: goto .L0`
	`5: label .L0`
	`6: proc_end add_func`

Table 5.35 shows a procedure invocation and its corresponding translated code generated using the productions and semantic actions discussed above. Figure 5.23 shows the attributes of different nodes and their values during the translation from the input source to the three address code. The nodes are numbered in the order of creation during bottom-up translation.

Table 5.35 *Input source and its translated code*

C code snippet	Translated TAC

`v3=add_func(v1,v2);`	`param v2`
	`param v1`
	`call add_func 8`
	`retrieve _t0`
	`v3 := _t0`

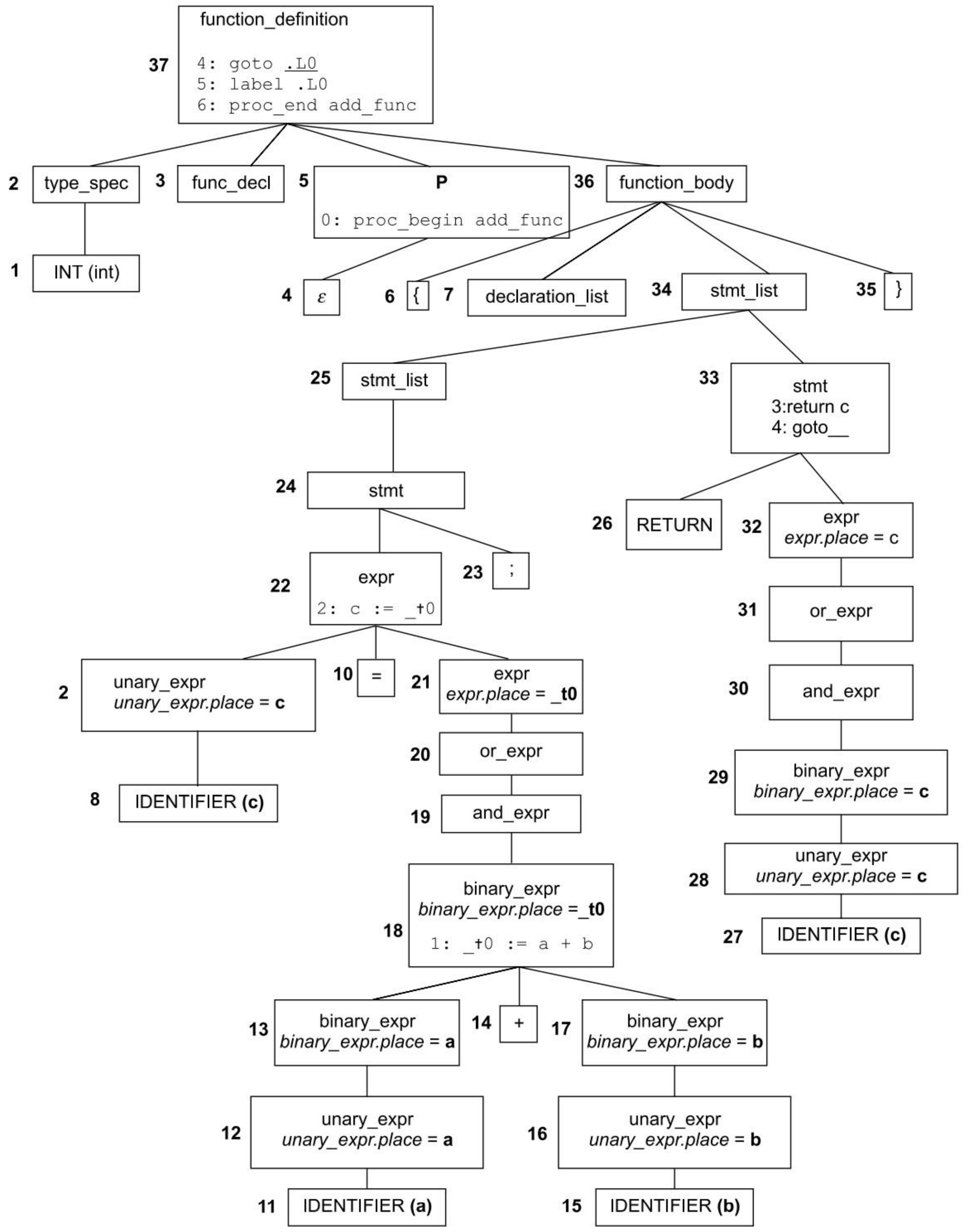


Fig. 5.22 *Translation of procedure definition*

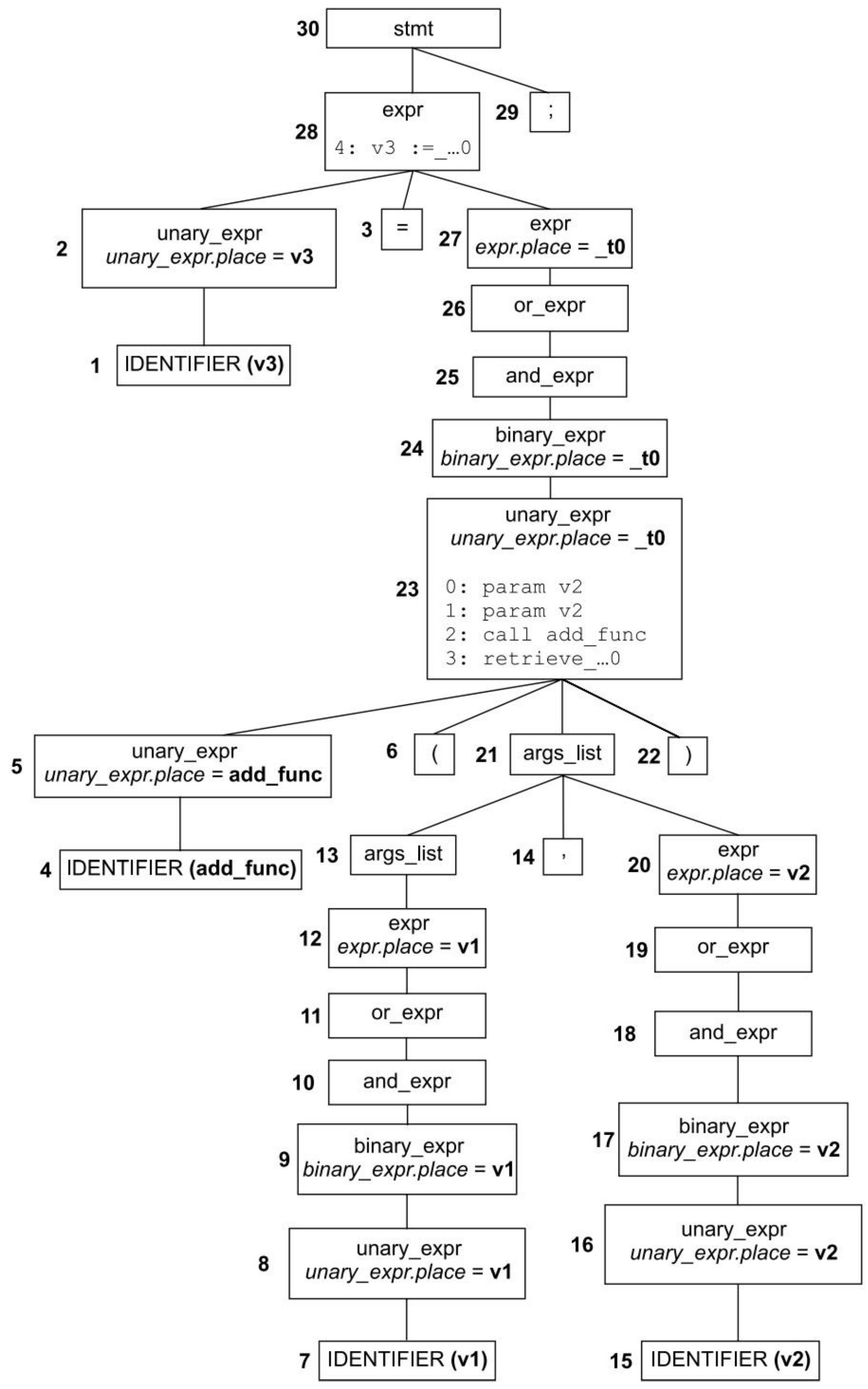


Fig. 5.23 *Translation of procedure invocation*

5.2.16 Example 8—Translation of Procedure Calls

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving procedure calls using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with statements involving procedure calls. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test8.c
     1  int z;
     2
     3  int add_func(int a,int b)
     4  {
     5       int c;
     6
     7       c = a + b;
     8
     9       return(c);
    10  }
    11
    12  int main()
    13  {
    14       int v1,v2,v3,v4;
    15
    16       v1=10;
    17       v2=20;
    18
    19       v3=add_func(v1,v2);
    20
    21       z=v3+5;
    22  }
    23
```

```
# Generating IC
$ ./icgen test8.c
(0) proc_begin add_func
(1) _t0 := a + b
(2) c := _t0
(3) return c
(4) goto .L0
(5) label .L0
(6) proc_end add_func

(0) proc_begin main
(1) v1 := 10
(2) v2 := 20
(3) param v2
(4) param v1
(5) call add_func 8
(6) retrieve _t0
(7) v3 := _t0
 (8) _t1 := v3 + 5
 (9) z := _t1
(10) label .L1
(11) proc_end main

# Input file
$ cat -n test8a.c
    1   int z;
    2
    3   int add_func(int a,int b)
    4   {
    5      int c;
    6
    7      c = a + b;
    8
    9      return(c);
   10   }
   11
   12   int main()
   13   {
   14      int v1,v2,v3,v4;
   15
   16      v1=10;
   17      v2=20;
   18
   19      v3=add_func((v1+6),v2);
   20
   21      z=v3+5;
   22   }
   23

# Generating IC
$ ./icgen test8a.c
(0) proc_begin add_func
(1) _t0 := a + b
(2) c := _t0
```

```
(3) return c
(4) goto .L0
(5) label .L0
(6) proc_end add_func

 (0) proc_begin main
 (1) v1 := 10
 (2) v2 := 20
 (3) _t0 := v1 + 6
 (4) param v2
 (5) param _t0
 (6) call add_func 8
 (7) retrieve _t1
 (8) v3 := _t1
 (9) _t2 := v3 + 5
(10) z := _t2
(11) label .L1
(12) proc_end main
```

SUMMARY

In a compiler, the front end translates the higher level language input source into a form that is independent of target machine architecture called as intermediate code. The intermediate code is simple enough to be mapped on to different target architectures. The main reason for generating intermediate code instead of the final target code is that it helps in easier retargeting of the compiler to generate instructions for different processors. There are a two different formats of intermediate code that were studied in this chapter, namely abstract syntax tree (AST) and three address code (TAC) format. The translation of input source into intermediate code is performed by using the syntax directed translation technique.

The production rules and the semantic actions of the translation scheme to generate intermediate code (TAC format) for common programming constructs encountered in higher-level languages were examined. The translation of declarations, which yields symbol table, was studied in the previous chapter. The generation of intermediate code for (1) Simple assignment statements; (2) Array, pointer and record references; (3) Flow of control statements (like if-else and while constructs); (4) Switch-case constructs; and (5) Procedure calls were covered in 5.2. The discussion on the translation of each of the programming construct was supplemented by a demonstration of IC generator program, generating the intermediate code for that specific programming construct. The generation of intermediate code for flow of control statements and some others presented a challenge of not knowing the target label to jump to at the time of emitting code. The backpatching technique helped us overcome this challenge and generate the intermediate code.

REVIEW QUESTIONS AND EXERCISES

5.1 A compiler can choose one of the two options (a) Translate the input source into intermediate code and then convert it to final machine code; (b) Directly generate the final machine code from the input source. What is the preferred option and why?

5.2 Describe the three address code form of the intermediate code. List out some of operators used in three address code with examples.

5.3 How can three address code be implemented in a compiler? Describe triples and indirect triples method of implementing TAC with examples.

5.4 Compare the different methods of implementing three address code.

5.5 How is an abstract syntax tree different from a parse tree? List out some of the nodes in the AST for a C compiler?

5.6 Translate a C statement '$a = b + c - (4*a*b + 3*c)$;' into TAC. How are the binary operators like +, –, etc., handled during the translation?

5.7 Translate an array reference statement '$a = b[c]$;' into TAC. What are the main TAC operators used during the translation? What attributes of a unary expression are used in translation of array references?

5.8 How is the offset calculated for a multidimensional array reference? Derive the formula.

5.9 Translate the C statements '*p*=& arr[3]; **p*=10;' into TAC. What TAC operators are useful during the translation of pointer accesses?

5.10 Translate the C statement '*x*.age = 30;' into TAC. Assume that the field 'age' is at an offset of 20 bytes from the base of the structure. What are the common TAC operators used during the translation of 'struct' references using the dot operator?

5.11 Translate the C statement 'ptr→age=20;'. Assume that the field 'age' is at an offset of 20 bytes from the base of the structure. What are the common TAC operators used during the translation of 'struct' references using the arrow operator?

5.12 Translate the C statement 'if (a<b){x=y;} m=20;' into TAC. In a single pass compiler, how is the translation of Boolean test expression ($a < b$) performed? How does it know about the labels to jump on being true or false?

5.13 Describe the backpatching technique. How is it used in the translation of an input C statement 'if ((a < b) || ($c < d$)) {m = 20;} else {m = 10;} $p = m$;'?

5.14 What are the data structures used during the translation of a 'while' statement? Illustrate the usage of those data structures during the translation of a C statement 'while ($i < b$){val = val **i*; $i = i + 1$;} m = val;'?

5.15 How is a switch-case statement translated into TAC? Illustrate with an example.

5.16 What are the calling and returning sequences? List out the TAC instructions generated during both of these sequences by taking a sample C code snippet.

5.17 What is the sequence of events in the called function during a procedure call? Illustrate with an example.

5.18 How is a call to a procedure translated into TAC? Illustrate with an example.

5.19 State if the following statements are true or false:

(a) The separation of a compiler into front end and back end is helpful in retargeting of the compiler.

(b) The separation of a compiler into front end and back end helps in adding support for a new source language easily.

(c) The intermediate code can be 'improved' by using techniques independent of the target architecture.
(d) The back end of the compiler takes the intermediate code to generate the target code.

5.20 State if the following statements are true or false:
(a) The backpatching technique is employed during the translation of array references.
(b) The 'place' and 'offset' attributes of a unary expression are used to help the translation of array, record references.
(c) The triples and indirect triples are methods of implementing three address code form of intermediate representation.
(d) The indirect triples implementation is more favourable to the optimisation of the TAC.

TARGET CODE GENERATION

6

Introduction

We studied in the previous chapters how the input source was broken up into tokens (Chapter 2—Lexical Analysis), verified against a specified grammar (Chapter 3—Syntax Analysis), checked for semantic errors (Chapter 4—Semantic Analysis) and translated to machine-independent intermediate code (Chapter 5—Intermediate Code Generation). In this chapter, we study about the conversion of the machine-independent intermediate code into target program, which is closer to the machine architecture. The conversion is accomplished by a ***target code generator*** (more simply called as code generator) that takes intermediate code as the input and generates a target program as output. The target program can be an assembly language program or absolute machine code.

The different forms of the target code and their advantages/ disadvantages is the topic of the discussion in Section 6.1. The intricacies of code generation are explored in this chapter by using x86 as a model target processor and its assembly language program as the target program. Section 6.2 gives an overview of the x86 processor and also its assembly language programming to help understand the target code generation better. The target code generator needs to conceive the run-time environment in which the target program runs and generate the target code accordingly. The important aspects of the run-time environment that the code generator needs to provide for are described in Section 6.3. The concepts of target code generation using a template-based approach are discussed in Section 6.4.

6.1 TARGET PROGRAM

The intermediate code, as we saw in the previous chapters, is a very generic set of instructions which is not restricted to any specific type of processor. The target program on the other hand, is a closer representation of the machine characteristics.

The target program can take one of the following three forms:

1. Assembly language program.
2. Absolute machine code.
3. Relocatable object code.

One of the advantages of choosing assembly instructions as the target program is that the target code generator can generate symbolic instructions (like mov *ax*, var1) instead of relocatable or absolute addresses (like mov *ax*, 0x80000000). This improves readability of the target code generated. It also makes it easier to debug the compiler during a malfunction. A target program in assembly language can also make use of the macro facilities of the assembler. The disadvantage of having the assembly language program as the target program is that an additional pass is required for converting it into machine instructions. In any case, multiple passes in the compiler is unavoidable because storing the entire set of data structures for translation from the input source to machine instructions would put a very high demand on the memory.

By having absolute machine code as the target program for code generation, there is a flexibility of loading the code in a fixed memory location and immediately executing it. However, this can work only if the program is small in size.

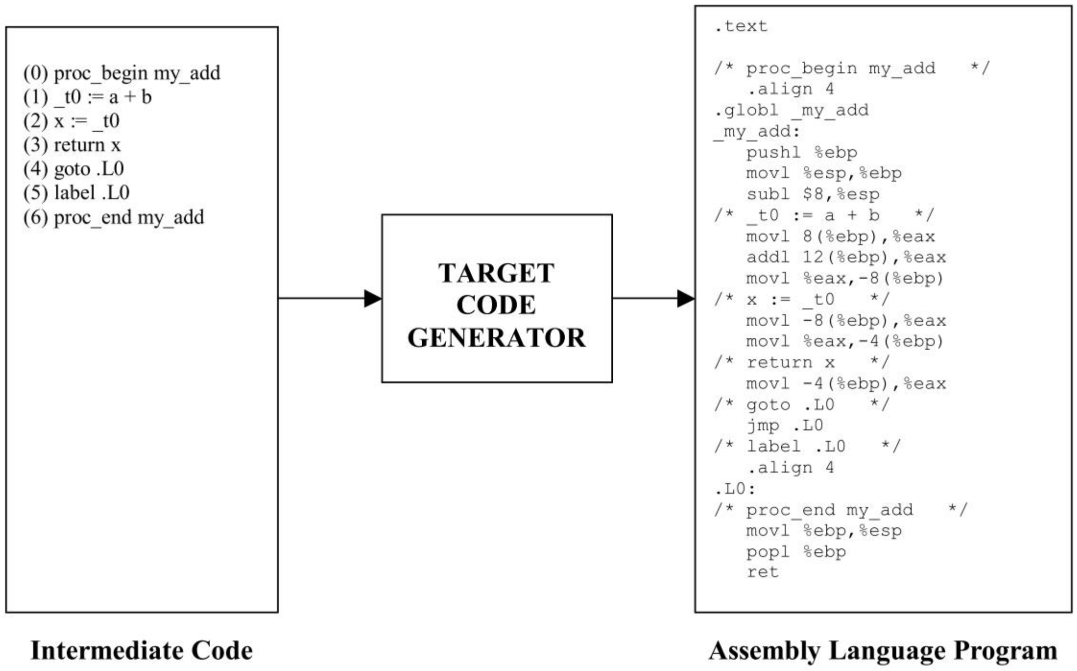
Having relocatable object code as a target program allows a program (say '*p*'), spread across multiple files (like say *p*1.*c*, *p*2.*c*, etc.) to be each compiled separately. The relocatable object modules (*p*1.*o*, *p*2.*o*, etc.) can be linked together and loaded for execution. This scheme allows the user to selectively compile a part of the program (like just *p1.c*) and link it together with previously compiled object modules to generate the executable.

Assembly language program is the target program of our choice for the code generator discussed in this chapter. This choice aligns well with most of the compilers that are commonly available. The target code generator discussed throughout this chapter takes the intermediate code as input and generates the assembly language program as the output. This is shown in Fig. 6.1.

The Intel's x86 family of processors is the model target processor for the study of code generation in this chapter. We examine the architecture of x86 family of processors and write some assembly language programs for it in the next section (Section 6.2). The knowledge of the x86 processor architecture and its assembly language programming would be of immense use during the discussion on target code generation for x86 in the later section (Section 6.4) of the chapter.

6.2 X86 PRIMER

In this section, we briefly study the architecture of the Intel's x86 processor, followed by some assembly language programming for the same. The idea here is to get acquainted with the processor details and write a few basic assembly programs. We cover the common x86 assembly language programming features like global variables, registers, arithmetic operations, pointers, and so on in the discussion here. For a more comprehensive treatment of the assembly language programming and the processor details, the reader should refer to the programming manual.



```
.text

/* proc_begin my_add   */
   .align 4
.globl _my_add
_my_add:
   pushl %ebp
   movl %esp,%ebp
   subl $8,%esp
/* _t0 := a + b   */
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-8(%ebp)
/* x := _t0   */
   movl -8(%ebp),%eax
   movl %eax,-4(%ebp)
/* return x   */
   movl -4(%ebp),%eax
/* goto .L0   */
   jmp .L0
/* label .L0   */
   .align 4
.L0:
/* proc_end my_add   */
   movl %ebp,%esp
   popl %ebp
   ret
```

Fig. 6.1 *Code generator translates intermediate code to assembly language program*

6.2.1 x86 Architecture

The Intel's x86 family of processors is one of the most widely used processors in the world. These processors form the core of the personal computers (PC) that we use at home. The x86 family of processors is based on CISC (complex instruction set computer) architecture. It supports a wide range of instructions and addressing modes.

The x86 family of processors has 8 general-purpose registers given by the names eax, ebx, ecx, edx, esp, ebp, esi and edi. The registers eax, ebx, ecx and edx are usually used for numerical computations. The registers esp, ebp, esi and edi are used for accessing stack and performing pointer manipulation. Table 6.1 shows the general purpose registers of x86 family and their typical usage.

Table 6.1 *General purpose registers*

Name	Description
EAX	Used in most of the arithmetic operations
EBX	Used in most of the arithmetic operations
ECX	Used in most of the arithmetic operations
EDX	Used in most of the arithmetic operations
ESP	Stack Pointer—Used in Stack manipulation
EBP	Base Pointer—Used in accessing elements on the stack from a reference point
ESI	Source Index—Used in moving chunks of data from source to destination memory
EDI	Destination Index—Used in moving chunks of data from source to destination memory

The 32 bit EAX, EBX, ECX and EDX registers can also be referenced by their 16bit and 8bit forms. For example, AX refers to the lower 16 bits of the register EAX. AH refers to the higher 8 bits of AX and AL refers to the lower 8 bits of AX. Figure 6.2 shows the relationship between the various forms.

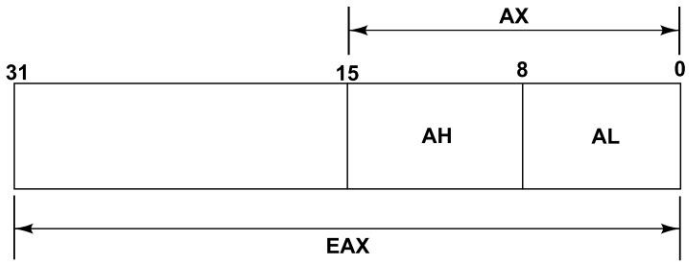


Fig. 6.2 *Register EAX and its representation*

There are 6 segment registers given by the names cs, ds, ss, es, fs and gs as shown in Table 6.2. The segment registers are used for accessing memory.

Table 6.2 *Segment registers*

Name	Description
CS	Code segment
SS	Stack segment
DS	A first data segment
ES	A second data segment
FS	A third data segment
GS	A fourth data segment

A ‘flags’ register is used to know the status of operations carried out by the processor. The flags register can tell if an overflow has happened during a multiplication or if a carry has been set during comparison and so on. The Instruction pointer register named IP gives the address of the next instruction to be executed.

Table 6.3 *Other registers*

Name	Description
IP	Instruction pointer
EFLAGS	Flags register used for knowing the status of operations performed by processor

6.2.2 Structure of an Assembly Language Program

The structure or the template of an assembly language program depends on the assembler used for translating it into Machine Code. In this chapter, we use the GNU’s assembler (commonly known as ***gas***) to convert the assembly language programs to machine code for the x86 processor. The assembly language programs that we would see in this chapter are all based on the format that GNU’s assembler expects. The ‘gas’ takes input assembly programs using AT&T Syntax. This syntax is somewhat different from the intel syntax that other assemblers like MASM follow.

There are three types of assembly language statements that can be seen in an assembly language program. They are

- Instructions
- Directives
- Macros

An ***instruction*** is translated by assembler into machine code that is executed at run-time. An assembly instruction can consist of 4 fields namely, the label, mnemonic to specify the instruction/size, operands on which the instruction operates, and a comment field beginning with a #. A sample assembly instruction illustrating all the four fields is shown below.

Label	**Mnemonic**	**Operands**	**Comment**
`l1 :`	`movl`	`%eax,var1`	`# Moving the contents of register eax into a variable var1`

A ***directive*** is used for informing the assembler to take some action. Directives do not result in machine code. All the assembler directive statements start with a dot (.). For example, the align directive is used for aligning the next assembly statement with 8 or 16 or 32 bit boundary. This directive helps in meeting the requirement of the processor regarding positioning of certain parts of the program at prescribed boundaries for efficiency. For example, in x86 architecture accessing a 16-bit value at addresses that are at multiples of 2 is more efficient than the addresses that are not. The align directive can be used to take advantage of this feature.

```
.align 2     #aligns the next assembly statement to 16 bit boundary
```

A ***macro*** is shorthand for a sequence of other statements. The assembler expands the macros to the statements it represents and then assembles it. For example, the following defines a macro called 'prolog'.

Table 6.4 *A macro definition*

```
.macro    PROLOG
       pushl %ebp
       movl %esp,%ebp
.endm
```

In Fig. 6.3 we can see that wherever the macro PROLOG is used, the assembler expands it and replaces it with the statements that it represents.

Assembly code before macro expansion	**Assembly code after macro expansion**
`.section .text`	`.section .text`
`.globl_my_add`	`.globl_my_add`
`_my_add:`	`_my_add:`
`PROLOG`	**`pushl %ebp`**
	`movl %esp,%ebp`
`movl _x,%eax`	`movl _x,%eax`
`addl _y,%eax`	`addl _y,%eax`
`movl %eax,_z`	`movl %eax,_z`

Fig. 6.3 *Expansion of macro by assembler*

An x86 assembly program would be typically divided into sections meant for specific purpose. The three sections used commonly are:

- Data section
- BSS (block started by symbol) section
- Text section

The data section is used for declaring the variables that have an explicit value to be initialised with. The BSS Section is used for declaring variables that are not initialised. The program loader usually initialises the BSS segment to a default value of 0. The text section defines the code (instructions) to be executed. The data and BSS sections are optional but the text section is mandatory. The '.section' directive is used for defining a section. Table 6.5 shows the three common sections defined in an assembly program.

Table 6.5 *Sections in an assembly program*

```
.section .data
     # Initialized Data
.section .bss
      # UnInitialized Data
.section .text
     # Instruction Code
```

A ***listing*** of assembly file can be used to view the assembly instructions side-by-side with the machine language instructions to which it was transformed. The listing is obtained by passing extra command line arguments to the assembler during assembly.

6.2.3 Assembly Language Programming

In this section, we take a quick look at some of the important features of assembly language programming for the x86 family using the GNU's assembler (gas). We try and understand the assembly language instructions by going through some code snippets and programs.

6.2.3.1 Global Variables and Arrays Global variables in an assembly program are commonly defined in data section. The following shows the declaration of an integer variable called 'my_var' having an initial value of 100.

```
# Declaration of a variable my_var with initial value as 100
my_var :
.int 100
```

As we can see, there are two statements used in defining a global variable, a label indicating the name of the variable and a directive signifying the size of the variable and its initial value. The comments are shown starting with the # mark.

Initialised global arrays are declared in the data section in a similar way with more number of data type (.int) directives. The following shows the declaration of a global array my_init_arr with 5 elements, each element being a 4-byte integer. The initial values for each of the five elements are 10, 20, 30, 40 and 50.

```
# Declaration of an integer global array my_init_arr with 5 elements
# The initial values for each of the five elements are 10,20,30,40 and 50

my_init_arr :
.int 10
.int 20
.int 30
.int 40
.int 50
```

Un-initialised global arrays in an assembly program are defined in bss section using the '.comm' (common memory) directive. The following shows the declaration of a 4-byte integer array called 'my_arr' having a size of 100 elements.

```
# Declaration of a 4-byte integer array my_arr with 100 elements (size =4x100= 400 Bytes)
.comm       my_arr,400
```

In the above case, the size of each element in the array is 4 bytes. There are 100 elements in it. This makes the size of the integer array 400 bytes. This size of 400 is used in the declaration using the '.comm' directive.

6.2.3.2 *Registers* The registers in x86 family of processors are accessed in assembly programs by prefixing the name with a % sign in front of them. For example, the content of eax register is accessed using %eax. The MOV instruction is commonly used for moving data across registers as well as memory. The MOV instruction has a suffix that indicates the number of bytes that are being moved from source to destination. A suffix of 'b' or 'w' or 'l' indicates movement of 1 byte or 2 bytes or 4 bytes respectively.

The 16 bit and 8 bit forms of the registers eax, ebx, ecx and edx similar to the one shown in Fig. 6.2 can also be accessed separately by using the corresponding reference. For example, %al refers to the lowest 8 bits in the register AX, %ah refers to the higher 8 bits of register AX. %ax refers to the 16-bit register AX. %eax refers to the complete 32-bit register EAX. The following shows the usage of various forms of the register.

```
movl %eax,%ebx     #moving 4 bytes of data from eax register to ebx register

movl %eax,var      #moving 4 bytes of data from eax register to label 'var'

movl var,%eax      #moving 4 bytes of data from variable 'var' to eax register

movw var,%ax       #moving 2 bytes of data from variable 'var' to ax register

movb var,%ah       #moving 1 byte of data from variable  'var' to ah register

movb var,%al       #moving 1 byte of data from variable  'var' to al register
```

6.2.3.3 *Immediate Operands* In assembly language programming using the AT&T syntax that we are following, the immediate operands are referenced using a $ sign in front of them. The following instruction shows the use of an immediate operand.

```
movl   $5,%eax       #moving a value 5 to eax register
```

6.2.3.4 *Arithmetic Operations* The x86 assembly language has instructions to perform the common arithmetic operations of add, subtract, multiply and divide.

The addition is performed by the ADD instruction whose syntax is given below.

add source, destination

where the source can be an immediate value, a memory location or a register. The destination can be a register or a memory location. This instruction adds the source value with the destination value and stores the result in the destination. Observe that the add instruction like many other instructions cannot operate when both source and destination are memory locations. As usual, the suffix of the instruction tells us the size of the operands (b for 8 bit, w for 16 bit, and l for 32 bit). The following are some of the examples of the ADD instruction.

```
addl %ecx,%eax      # Adds the content of register ecx to that of register eax
                    # and the result stored in register eax

addl var,%eax       # Adds the 4 bytes of data at label 'var' with content
                    # of the register eax.
                    # The result stored in register eax

addl %eax,var       # Adds the content of the register eax with 4 bytes of
                    # data at label 'var' and the result stored at the label var
```

The subtraction is performed by the SUB instruction whose syntax is given below.

sub *source, destination*

where the source can be an immediate value, a memory location or a register. The destination can be a register or a memory location. This instruction subtracts the source value from the destination value and stores the result in the destination. The sub-instruction cannot operate when both source and destination are memory locations. The suffix of the instruction tells us the size of the operands (b for 8 bit, w for 16 bit, and l for 32 bit). The following are some of the examples of the SUB instruction.

```
subl %ecx,%eax       # Subtracts the content of register ecx from that of register
                     # eax and the result stored in register eax

subl var,%eax        # Subtracts the 4 bytes of data at label 'var' from the
                     # content of the register eax and the result stored in
                     # register eax

subl %eax,var        # Subtracts the content of the register eax from 4 bytes
                     # of data at label 'var' and the result stored
                     # at the label var
```

The multiplication is performed by the MUL instruction whose syntax is given below.

mul *source*

where the source can be a memory location or a register. The destination operand is implied and it is a form of AX register depending on the suffix of the mnemonic. The destination operand is AL for the mnemonic mulb, AX for mulw, EAX for mull. This instruction multiplies the source value with the destination value and stores the result in AX for mulb, DX:AX for mulw, EDX:EAX for the mull. For the mulw and mull, the higher bits are stored in the DX or EDX respectively. The following are some of the examples of the MUL instruction.

```
mull %ecx      # Multiplies the content of register ecx with that of
               # register eax and the result stored in register
               # combination edx:eax

mull var       # Multiplies the 4 bytes of data at label 'var'
               # with the content of the register eax and the result stored
               # in register combination edx:eax

mulb %cl       # Multiplies the 8 bit content of the register cl with
               # the 8 bit content of al
               # and the result stored in register ax
```

The MUL instruction allows us to multiply unsigned integers. The IMUL instruction allows us to multiply signed integers. The IMUL instruction operates exactly like the MUL excepting that it interprets the sign by using the higher bits. The format, source operand, destination operand and the location where the result is stored remain the same.

The division is performed by the DIV instruction whose syntax is given below.

div *divisor*

where the divisor can be a memory location or a register. The dividend is implied and it is a form of AX register depending on the suffix of the mnemonic. The dividend is AX for the mnemonic divb, DX:AX for divw, EDX:EAX for divl. This instruction divides the implied dividend by the divisor and stores the result(quotient) in AL for mulb, AX for mulw, and EAX for the mull. The remainder of the division is stored in AH, DX and EDX for mulb, mulw and mull respectively. The following are some of the examples of the DIV instruction.

```
divl %ecx         # Divides the content of register pair edx:eax with
                  # the content of register ecx and the quotient
                  # is stored in register eax, the remainder in edx

divl var          # Divides the content of register pair edx:eax with
                  # the 32 bit value at label var and the quotient
                  # is stored in register eax, the remainder in edx

divb %cl          # Divides the content of register ax with the content
                  # in cl and the quotient is stored in register al,
                  # the remainder in dl
```

The DIV instruction allows us to divide unsigned integers. The IDIV instruction allows us to divide signed integers. The IDIV instruction operates exactly like the DIV excepting that it interprets the sign by using the higher bits. The format, source operand, destination operand and the location where the result is stored remain the same.

6.2.3.5 *Addresses and Pointers*

We have seen earlier that registers can contain data. They can also be used for containing memory addresses. A register containing memory address is usually referred to as a pointer. Using the memory address in the register, it is possible to access the memory location. This is called as ***indirect addressing***.

We can store the address of a variable in a register using the assembly instruction lea—load effective address. The syntax of the lea instruction is as follows.

lea source, destination

The source points to a memory location like a label. The destination is a 32-bit register where the address of the source object will be stored. An example of lea instruction is shown below.

```
leal var, %esi         # Loads the effective address of the variable/label var
                       # into register esi
```

The other way of getting the address is to use the $ sign as a prefix to the variable. An example is shown below.

```
movl $var, %esi           # Loads the effective address of the variable/label var
                          # into register esi
```

In order to access memory location using indirect addressing, the braces, '(' and ')' are used in conjunction with the register name. For example (%esi) is used for accessing the location pointed to by esi. Some assembly instructions using the indirect addressing are shown below.

```
movb $100, (%esi)    # Moves 100 into the location pointed by the register esi

movb (%esi),%eax     # Moves the content of location pointed by the register esi
                     into register eax
```

Using the indirect addressing, it is possible to access memory at a given offset from the location pointed to. For example, 4(%esi) can be used to access a memory location that is 4 bytes *after* the location pointed to by the register esi. A negative offset can be used to refer to the memory *before* the location pointed to. For example, –4(%esi) refers to a location that is 4 bytes before the location pointed to by the register esi. Some examples of assembly instructions using the indirect addressing with offsets are shown below.

```
movb $100, 6(%esi)     # Moves 100 into the location which is 6 bytes after
                       # the memory pointed to by the register esi

movl 10(%esi), %eax    # This moves the data that are at 10,11,12 and 13 byte offsets
                       # from the pointer esi into register eax

movb $100,  -10(%esi)  # This moves the data that are at 7,8,9 and 10 byte offsets from
                       # the pointer esi into register eax.
```

Figure 6.4 shows the concept of indirect addressing discussed above.

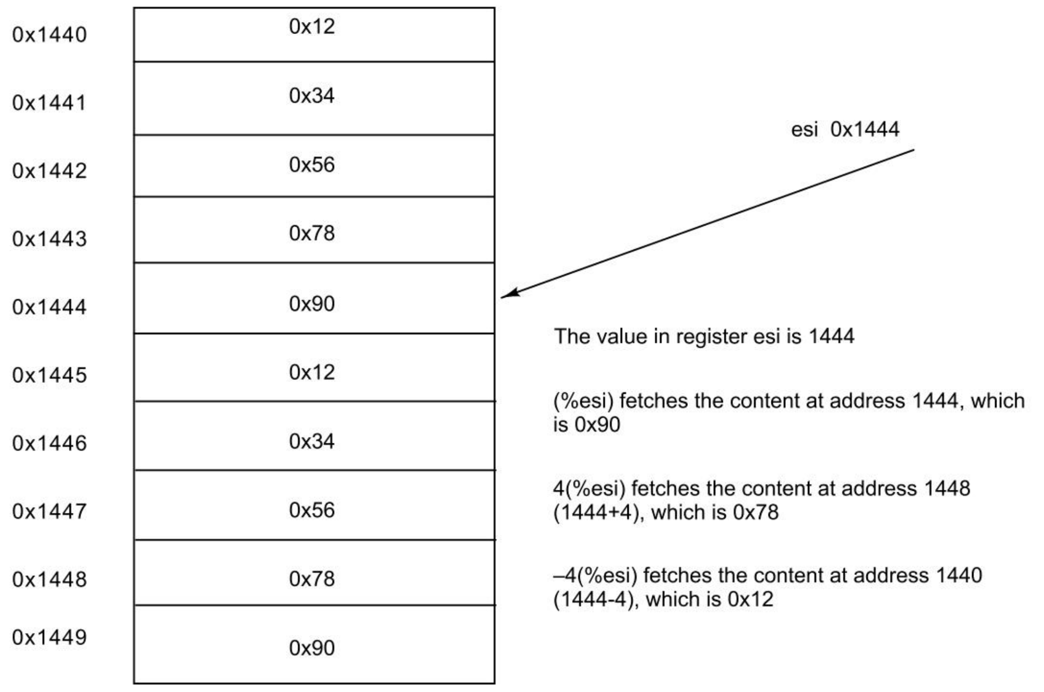

0x1440
0x12
0x1441
0x34
0x1442
0x56
0x1443
0x78
0x1444
0x90
0x1445
0x12
0x1446
0x34
0x1447
0x56
0x1448
0x78
0x1449
0x90
esi 0x1444
The value in register esi is 1444
(%esi) fetches the content at address 1444, which is 0x90
4(%esi) fetches the content at address 1448 (1444+4), which is 0x78
–4(%esi) fetches the content at address 1440 (1444-4), which is 0x12

Fig. 6.4 *Indirect addressing*

6.2.3.6 *Functions* A function is a logical entity to achieve a specific purpose. An assembly function begins with a label, which is the same as the name of the function. For example, a function called my_func begins with a label my_func. The first instruction following the label is the start of the function. The 'ret' instruction signifies the end of a function. A template for defining a function is shown below.

```
my_func :
     # The function Body come here
     ret
```

Fig. 6.5 *A Template for an assembly function*

The assembly instructions in the function body can access any global variables and registers. Listing 6.1 shows a part of assembly program ex1.s (assembly file names usually have a .s extension) that defines a function, which adds two global variables *x* and *y* and stores the result in another global *z*.

```
.data
x:
.int 10
y:
.int 20
z:
.int 0

.text

# The function 'my_add' adds the values in the global
# variables x and y and stores the result in global z.

my_add :

    # Function Body Begins
    movl  x,%eax
    addl  y,%eax
    movl  %eax,z
    # Function Body ends

    ret

```

Listing 6.1 *ex1.s*

A function can be invoked by a 'call' assembly instruction. The call instruction pushes the return address on the stack and transfers the control to the function. The '_main' is a special global function that serves as the default entry point for the program. Listing 6.2 shows the continuation of assembly program ex1.s in which the 'call' instruction is used to invoke the 'my_add' function defined earlier.

```
 .globl _main
_main :
    call my_add

    mov $0,%eax # returning 0 to keep the OS happy

    ret

.end
```

Listing 6.2 *Continuation of ex1.s*

The following dialog shows how to assemble the program ‘ex1.s’ and generate a binary that can be executed.

```
# Invoking the assembler to create an object file
# ..and linking it with startup file (crt0.o) to make the executable
$ gcc -g ex1.s -o ex1

# Executing it
$ ./ex1.exe
```

The common C Library functions (e.g. scanf, printf, strlen, etc.) can also be called by using the ‘call’ instruction. When a C library function is called using the ‘call’ instruction, the function name needs to be prefixed with an underscore (_) sign. For example, to call the C library printf function, the assembly instruction ‘call _printf’ should be used.

When the function ‘my_add’ was called in the earlier example, there were no arguments passed to it. Let’s now learn how to pass the arguments to a function when it is called using the ‘call’ assembly instruction. The following steps are involved in invoking a function with arguments.

- The arguments to a function are pushed on the stack using the ‘push’ instruction. While calling C library functions, the order in which the arguments are pushed on the stack is last to first, i.e. the last argument is pushed first, the last but one, next and so on. The first argument is the last one to be pushed on the stack.
- The function is invoked using the ‘call’ instruction.
- After the called routine returns, it is the caller’s responsibility to remove the arguments from the stack. This can be done by a series of pop instructions or by simply incrementing the SP register with the number of bytes pushed.

The Listing 6.3 shows an assembly program which invokes the C library routine printf, first with one argument and next with two arguments to clarify the function calling conventions mentioned above.

```

.data
year :
.int 2005

str1 :
.ascii “Hello World \n\0”

str2 :
.ascii “Hello World %d \n\0”

.text

```

```
 .globl _main

_main:

   # Equivalant of doing printf(“Hello World \n”);

   pushl $str1     # Push the address of the str1 as argument
   call _printf    # Calling the C Library Function - printf
   addl $4,%esp    # Reset the SP to ‘remove’ the pushed items

   # Equivalant of doing printf(“Hello World %d \n”,year);

   pushl year      # push the Last argument first
   pushl $str2     # Pushing the Last but one (first) argument
   call _printf    # Calling the C Library Function - printf
   addl $8,%esp    # Reset the SP to ‘remove’ the pushed items

   mov $0,%eax     # Return 0 to keep the OS Happy

   ret

.end
```

Listing 6.3 *ex1a.s*

The following dialog shows the assembling, linking and execution of example 1a.

```
# Invoking the assembler to create an object  le
# ..and linking it with startup  le (crt0.o) to make the executable
$ gcc -g ex1a.s -o ex1a

# Executing it
$ ./ex1a.exe
Hello World
Hello World 2005
```

How do we fetch the return value of a function? The convention is that, if the return value is less than or equal to 32-bit value, then one of the forms of the register EAX is used to get the return value. The 8-bit return values are fetched from register ‘al’. The 16-bit return values from register ax, and the 32-bit return values from register eax. For 64-bit return values, the higher 32 bits are in edx, and the lower 32 bits are in eax.

Let’s check out this convention by the program ex1b.c (Listing 6.4) in which we find the length of the string ‘Hello World’ using the C library function ‘strlen’ and then display it.

```
.data

ret_val :
.int 0

str1 :
.ascii “Hello World\0”
```

```

str2 :
.ascii "Return value of strlen is %d \n\0"

.text
  .globl _main

.align 4
_main:

      # Equivalant of doing ret_val = strlen(str1);

      pushl $str1          # Push the address of the str1 as argument
      call _strlen         # Calling the C Library Function - strlen
      movl %eax,ret_val    # Moving the return value into variable ret_val
      addl $4,%esp         # Reset the SP to 'remove' the pushed argument

      # Displaying the return value by using the equivalant of
      # printf("Return value of strlen is %d \n",ret_val);

      pushl ret_val        # push the Last argument first
      pushl $str2          # Pushing the Last but one (first) argument
      call _printf         # Calling the C Library Function - printf
      addl $8,%esp         # Reset the SP to 'remove' the pushed arguments

      mov $0,%eax          # Returning 0 to keep the OS happy

      ret

.end
```

Listing 6.4 *ex1b.s*

The following dialog shows the details of assembling, linking and execution of the example ex1b.c.

```
# Invoking the assembler to create an object file
# ..and linking it with startup file (crt0.o) to make the executable
$ gcc -g ex1b.s -o ex1b

# Executing it
$ ./ex1b.exe
Return value of strlen is 11
```

We learnt about calling functions (with and without arguments) and also fetching their return values. We have also learnt to define functions that do not have any arguments passed to it or return any value (ex1.s). We now turn to defining functions that have arguments passed to it and have return values.

As we have already seen, the arguments to the function are passed on the stack. Let's rework the function my_add, seen earlier in Example 1, to take two numbers as arguments and have their sum as the return value. From the conventions that we studied in the previous examples, the calling part would be as shown below.

```
# Pushing the Two arguments for my_add(x,y)
pushl y
pushl x

# Invoking my_add function
call my_add

# Fetching the return value of my_add function
movl %eax, z

# Restoring the stack to remove the pushed arguments
addl $8, % esp
```

The function definition for my_add requires a fuller understanding of the runtime settings. At the time of entry into the function my_add, the stack is as shown in Fig. 6.6. The two arguments are on the stack owing to the pushes made before calling the function. The return address is on the stack because of the 'call' assembly instruction, which pushes it on to the stack before transferring the control to the my_add function.

From Fig. 6.6, it is clear that the function arguments can be accessed in the body of the function by using indirect addressing via the ESP. For example, if we assume the return address, argument 1 and argument 2 are all 4 bytes each, the function argument 1 can be accessed as 4(%ESP), argument 2 as 8(%ESP) and so on. The pitfall in accessing the arguments in indirect addressing via SP is that, if the SP gets changed in the function body due to a push, then the ability to access the arguments is completely lost. A more safer approach is to move the value of esp into ebp at the time of entry of the function, so that all the arguments in the function body can be accessed by indirect addressing via ebp like 4(%ebp), 8(%ebp) and so on. The only restriction is that the bp should not get changed in the function, which is reasonable. The additional aspect that needs to be taken care of is to save the BP value at the entry of function and restore it back at the time of exiting the function.

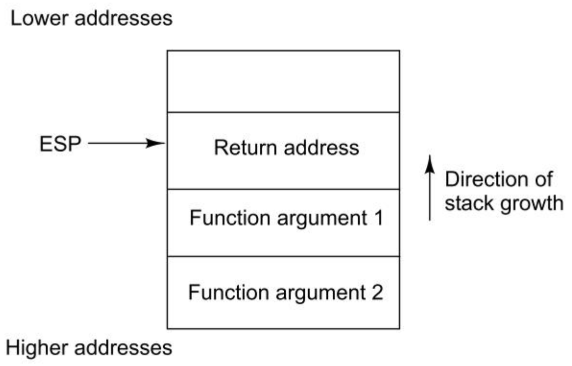


Fig. 6.6 *Stack at the entry point of a function*

Assuming that each of the function argument is a 4-byte value, Fig. 6.7 shows the stack as seen in the body of function. The function arguments are accessed by indirect addressing via the bp.

The first argument is at 8(%ebp), the second at 12(%ebp), and so on. For arguments, which are not 32-bit values, the calculations should take into account the size of the arguments. The first argument is at 8(%ebp), while the second argument is located at offset (%ebp), where offset = 8+sizeof first argument. The third argument

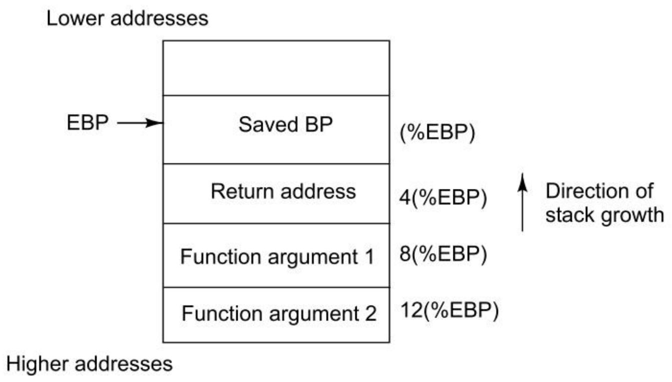


Fig. 6.7 *Function arguments accessed by indirect addressing via BP*

is located at an offset = 8 + size of first argument + size of second argument and so on. The alignment that the processor is particular about is that the size of each argument should be multiples of 4.

In order to support function arguments, the following two guidelines need to be adhered to while defining a function.

- A ***prolog code*** in which the current value of BP is saved and then the SP's value is moved to BP, should be added at the entry of each function.
- An ***epilog code*** in which the SP and BP are restored to their respective original values at the time of function entry, should be added at the function exit.

Figure 6.8 shows the modified function template including the prolog and epilog.

```
my_func:
      # function Prolog begins

      pushl    %ebp           # Saving the current BP
      movl     %esp,%ebp      # Moving the SP to BP so that
                              # function arguments can be accessed using
                              # indirect addressing via BP

      # Function Prolog ends

      # The function Body begins

      # The Function Arguments can be accessed here using indirect addressing via BP
      # for e.g. movl 4(%ebp), %eax which moves the content of Function Argument 1
      # into register eax

      # The function Body ends

      # Function Epilog begins

       movl    %ebp,%esp      # Restoring the sp to what it was at the entry of function
       popl    %ebp           # Restoring the value of BP

      # Function Epilog Ends

      ret
```

Fig. 6.8 *Assembly function template including prolog and epilog*

In order to return a value, the convention is that the called function moves the return value to some form of the register EAX and returns back to the caller. 8-bit values are returned in al, 16-bit values are returned in ax and the 32-bit values are returned in eax. It is the caller's responsibility to retrieve the return value from the appropriate form of EAX and use it.

Listing 6.5 shows the reworked function my_add, seen earlier in Example 1, to take two numbers as arguments and have their sum as the return value.

```

.data
x:
.int 10
y:
```

```
.int 20
z:
.int 0

str2 :
.ascii "The Sum of two numbers x=%d y=%d is z=%d \n"

.text

# The function 'my_add' adds the values in the global variables x and y
# and stores the result in another global z.

my_add :

   # Function Body Begins
   pushl  %ebp             # Saving the current BP
   movl   %esp, %ebp       # Moving the SP to BP so that
                           # function arguments can be accesed using
                           # indirect addressing via BP

   movl   8(%ebp),%ecx     # Moving the First Argument to %ecx
   addl   12(%ebp),%ecx    # Adding the second argument to the
                           # content of %ecx and store it in %ecx


  movl    %ecx,%eax        # The return value has to be stored in %eax

  movl    %ebp,%esp        # Restoring the SP to what it was at the
                           # entry of function
  popl    %ebp             # Restoring the value of BP

                           # Function Body ends

  ret

 .globl _main
_main :

  # Pushing the Two arguments for my_add(x,y)
  pushl y
  pushl x
  call my_add
  movl %eax,z
  addl $8,%esp

  # Displaying the return value by using the equivalant of
  # printf("The Sum of two numbers x=%d y=%d is z=%d \n",x,y,z);

  pushl z              # push the Last argument first
  pushl y              # push the Last but one argument next
  pushl x              # push the next argument in the reverse order
  pushl $str2          # Pushing the Last but one (first) argument
  call _printf         # Calling the C Library Function - printf
  addl $16,%esp        # Reset the SP to 'remove' the pushed arguments

  mov $0,%eax        # returning 0 to keep the OS happy


  ret

.end
```

Listing 6.5 *ex1c.s*

The dialog below shows the details of assembling, linking and executing the Example 1c.

```
# Invoking the assembler to create an object file
# ..and linking it with startup file (crt0.o) to make the executable
$ gcc -g ex1c.s -o ex1c

# Executing it
$ ./ex1c.exe
The Sum of two numbers x=10 y=20 is z=30
```

The local variables are also allocated on the stack. The register ebp is used to access the local variables via the indirect addressing similar to the function arguments. While the function arguments are accessed using positive offsets, the local variables are accessed using negative offsets from ebp as shown in Fig. 6.9.

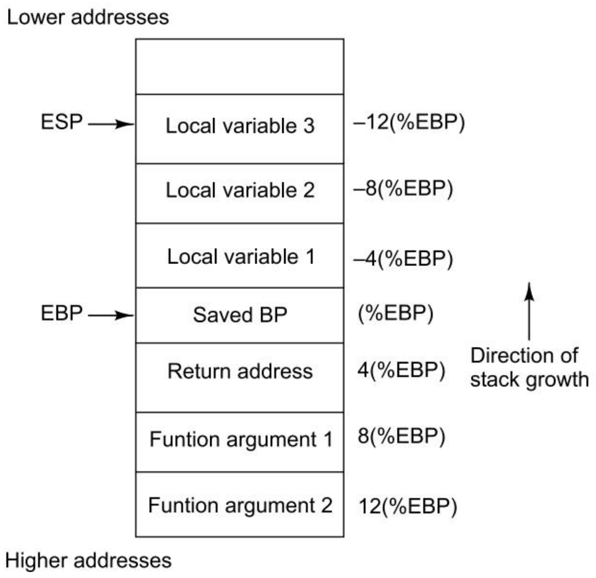


Fig. 6.9 *Stack showing local variables and function arguments*

For using local variables in an assembly language routine, the following procedure needs to be adhered to:

- The storage space is allocated for the local variables at the start of the function. This is done through decrementing SP by the amount of space required by the local variables. For example, if there are 2 local variables (each is say a 4-byte integer), then the SP is decremented by 8. The first local variable would constitute the 4 bytes –1(%ebp), –2(%ebp), –3(%ebp) and –4(%ebp). The second local variable would constitute the 4 bytes starting at –5(%ebp) and extending till –8(%ebp).
- The local variables are accessed in assembly instructions with indirect addressing via EBP register. The following instructions show sample local variable accesses.

```
movl -4(%ebp),%eax #moving the value of local variable 1 into eax
movl $5, -8(%ebp)  #Assigning 5 to the Local variable 2
```

- The local storage space is reclaimed just before exiting the assembly routine. This is done by restoring the SP to its value at the entry of the function (before it was decremented to create local space). This is achieved by simply moving the current value of BP into SP, since BP is currently pointing to the place where SP was at the start of function (see Fig. 6.6). This functionality is anyway part of the epilog of a function that we studied earlier.

The template of the function using function arguments as well as local variables and returning a value is shown below.

```
my_func:

     # Function Prolog begins

     pushl   %ebp           # Saving the current BP
     movl    %esp,%ebp      # Moving the SP to BP so that
                            # function arguments can be accessed using
                            # indirect addressing via BP

     # Creating Storage for Local Variables
     subl    $12,%esp       # For a function needing 12 Bytes of Local storage

     # Function Prolog Ends

     # The function Body begins
     # The Function Arguments can be accesed here using indirect addressing via BP
     # for e.g.movl 4(%ebp), %eax which moves the content of Function Argument 1
     # into register eax

     # The Local Variables can be accessed using indirect addressing via BP
     # for e.g.movl –4(%ebp), %eax which moves the content of local variable 1
     # into register eax

     # If there is a return value for the function, move the return value
     # to some form of AX register. AL for 1 byte return value,
     # AX for 2 byte return value, EAX for 4 byte return value

     # The function Body ends

     # Function Epilog begins

     movl    %ebp,%esp      # Restoring the SP to what it was at the
                            # entry of function
     popl    %ebp           # Restoring the value of BP

     # Function Epilog Ends

     ret
```

Fig. 6.10 *Template for function capable of handling arguments and local variables*

The Example 1d in Listing 6.6, shows a program in which there is a function swap, which handles arguments, returns a value and also has local variables.

```

.data
x:
.int 10
y:
```

```
.int 20

str1 :
.ascii "The two numbers before swap x=%d y=%d \n\0"

str2 :
.ascii "The two numbers after swap x=%d y=%d \n\0"

.text

# The function 'swap' swaps the values in the global variables x and y
# by using a temporary local variable - tmp on the stack

swap :

   # Function Prolog begins
   pushl   %ebp            # Saving the current BP
   movl    %esp,%ebp       # Moving the SP to BP so that
                           # function arguments can be accessed using
                           # indirect addressing via BP

   # Creating local storage space
   subl    $4,%esp # Creating 4 bytes of Local storage

   # Function Prolog ends

   # Function Body Begins

   # tmp = x ;
   movl    x,%eax          # Moving x into register eax
   movl    %eax,-4(%ebp)   # Moving it into local variable (tmp)

   # x = y ;
   movl    y,%eax          # Moving y into register x
   movl    %eax,x          # Moving it into x

   # y = tmp ;
   movl    -4(%ebp),%eax   # Moving local variable (tmp) to reg x
   movl    %eax,y          # Moving reg x into y

   # Function Body ends


   # Function Epilog begins
   movl    %ebp,%esp       # Restoring the SP to what it was at the
                           # entry of function
   popl    %ebp            # Restoring the value of BP

   # Function Epilog ends

   ret

  .globl _main
_main :

   # printf("The two numbers before swap x=%d y=%d \n",x,y);
   pushl y                  # push the Last argument
   pushl x                  # push the next argument in the reverse order
   pushl $str1              # Pushing the Last but one (first) argument
   call _printf             # Calling the C Library Function - printf
   addl $12,%esp            # Reset the SP to 'remove' the pushed arguments

   call swap                # Calling the swap function


   # printf("The two numbers after swap x=%d y=%d \n",x,y);
   pushl y                  # push the Last argument
   pushl x                  # push the next argument in the reverse order
   pushl $str2              # Pushing the Last but one (first) argument
   call _printf             # Calling the C Library Function - printf
   addl $12,%esp            # Reset the SP to 'remove' the pushed arguments


   mov $0,%eax              # returning 0 to keep the OS happy

   ret

.end
```

Listing 6.6 *ex1d.s*

The following shows the dialog for assembling, linking and executing Example 1d.

```
# Invoking the assembler to create an object file
# ..and linking it with startup file (crt0.o) to make the executable
$ gcc -g ex1d.s -o ex1d

# Executing it
$ ./ex1d.exe
The two numbers before swap x=10 y=20
The two numbers after swap x=20 y=10
```

6.2.3.7 Decision-making and Jumps In the normal course of a program, the instructions get executed sequentially one after the other. However, programs often require branching out to different parts of code depending on certain conditions. For example, a program might invoke a function A or function B depending on whether a variable 'g' is greater than 50 or less than 50. This kind of conditional jumps are an important part of programming.

At an assembly instruction level, the conditional jumps happen in two steps:

1. Comparing two values.
2. Jumping to a label depending on the result of comparison.

The comparison of two values is performed using the 'cmp' assembly instruction. The syntax of the cmp instruction is as follows.

cmp operand1,operand2

This compares operand2 with operand1. The operands can be registers or immediate values or a memory location (both cannot be memory locations at the same time). The operands are not modified by the instruction, but the status flags in EFLAGS register are set.

Jumping to a label depending on the content of EFLAGS register is performed by the conditional jump instruction, which has the following syntax.

jxx address

where the *xx* is a one- or two- or three-letter code for the condition and address is the target label to jump.

An example of conditional jump instruction is 'ja' instruction, which stands for jump if above. The ja instruction checks the two bits CF (Carry Flag) and ZF (zero flag) in the EFLAGS register to make the decision as to jump to the label or not. The following shows some examples of jumps to labels depending on the result of comparisons.

```
cmpl   %eax, $100      # Compare 100 with the content in register eax
jl     my_lbl          # Jump to my_lbl if 100 is less than content of
                       # register eax

cmpl   %eax,  $100     # Compare 100 with the content in register eax
jle    my_lbl          # Jump to my_lbl if 100 is less or equal to
                       # content of register eax

cmpl   %ecx,%eax       # Compare the content of register eax with that of ecx
jne    my_lbl          # Jump to my_lbl , if they are not equal
```

A complete list of all the conditional jump instructions like ja, jle, etc. can be found in an x86 assembly language programming manual.

The Example 1e in Listing 6.7, shows a program in which we use conditional jump feature in the x86 assembly language.

```

.data
x:
.int 10
y:
.int 20
z:
.int 30

str1 :
.ascii "The numbers before compare x=%d y=%d z=%d \n\0"

str2 :
.ascii "The numbers after compare x=%d y=%d z=%d \n\0"

.text

 .globl _main
_main :

   # printf("The numbers before compare x=%d y=%d z=%d \n",x,y,z);
```

```
  pushl z             # push the Last argument
  pushl y             # push the Last but one argument
  pushl x             # push the next argument in the reverse order
  pushl $str1         # Pushing the first argument
  call _printf        # Calling the C Library Function - printf
  addl $16,%esp       # Reset the SP to 'remove' the pushed arguments

  movl    x,%eax       # Moving x into reg eax
  cmpl    %eax,y       # Comparing y with x
  jl      lbl1         # jump to lbl1, if y is less than x

  movl    $50,z        # move 50 into z, if y is >= x
  jmp     lbl3         # jumping to lbl3

lbl1:
  movl    $100,z       # move 100 into z, if x is < y

lbl3:

  # printf("The numbers after compare x=%d y=%d z=%d \n",x,y,z);
  pushl z              # push the Last argument
  pushl y              # push the Last but one argument
  pushl x              # push the next argument in the reverse order
  pushl $str2          # Pushing the first argument
  call _printf         # Calling the C Library Function - printf
  addl $16,%esp        # Reset the SP to 'remove' the pushed arguments


 mov $0,%eax           # returning 0 to keep the OS happy

ret

.end
```

Listing 6.7 *ex1e.s*

The following shows the dialog for assembling, linking and executing example 1e.

```
# Invoking the assembler to create an object file
# ..and linking it with startup file (crt0.o) to make the executable
$ gcc -g ex1e.s -o ex1e

# Executing it
$ ./ex1e.exe
The numbers before compare x=10 y=20 z=30
The numbers after compare x=10 y=20 z=50
```

We have discussed about some of the important aspects of x86 assembly programming in this section. The grasp on the essentials of the x86 assembly programming is very vital for understanding the principles of code generation that would be studied in a later section. The code generator like all the assembly programs that we saw in this section uses AT&T syntax for the generated x86 assembly code.

6.3 RUNTIME ENVIRONMENT

During the target code generation, the target code generator needs to consider what kind of runtime settings or ***runtime environment*** should be provided for the execution of program. 'Runtime environment' is a term used to broadly describe all the runtime settings. In the case of compiled languages the runtime environment is indirectly controlled by generating the code to maintain it. In the case of an interpreted program like, say, a PERL script, the runtime environment is maintained directly in the data structures of the interpreter (PERL).

A compiler designer should conceive the environment under which a program is expected to execute and have the code generator generate the target code accordingly. In this section we discuss the various aspects of a runtime environment and how the target code needs to be generated in order to implement the environment. We start off with a discussion on the terminology used in describing runtime environment in Section 6.3.1. This is followed by discussion on the important elements of a runtime environment. The discussion is supplemented by taking examples of how certain features are implemented in the runtime environments used in C language, Pascal and FORTRAN77 compilers.

6.3.1 Terminology

When a program is executed, the control flows sequentially instruction after instruction. When a procedure is called, the control is transferred to the first instruction in the procedure. After executing all the instructions in the procedure, the control returns back to the location where the procedure was called. The program continues the execution with the next instruction after the call.

Strictly speaking, a ***function*** is a procedure, but with a return value. However, we use the terms procedure and functions interchangeably throughout this chapter for the sake of convenience.

The execution of a procedure is called as the ***activation*** of the procedure. The ***lifetime*** of a procedure is the time spent in execution of a procedure, including the time spent in other procedures called by it. The flow of control in a program can be depicted by an ***activation tree***. In an activation tree, each of the function activation represents a node. Figure 6.11 shows an activation tree corresponding to the execution of a string reversal C program shown in the dialog below.

```
# String Reversal Program in C
$ cat -n ex2.c
     1  #include <stdio.h>
     2  #include <string.h>
     3
     4  char str[100]="Compiler";
     5
     6  void string_reverse(char *a,int b);
     7  void swap(int a,int b);
     8  void my_print(char c1,char c2);
     9
    10  int main()
    11  {
    12          int len;
    13
    14          len=strlen(str);
    15          printf("%s\n",str);
    16          string_reverse(str,len);
    17          printf("%s\n",str);
    18          return(0);
}

void string_reverse(char *c,int len)
{
        int i,mid,l1;

        l1=(len-1);
        mid=l1/2;

        for(i=0;i<=mid;i++){
             swap(i,(l1-i));
        }
}

void swap(int a,int b)
{
        char tmp,c1,c2;

        c1=str[a];
        c2=str[b];

        my_print(c1,c2);

        tmp=str[a];
        str[a]=str[b];
        str[b]=tmp;
}

void my_print(char c1,char c2)
{
        printf("Swapping %c with %c\n",c1,c2);
}

```

```
# Compiling it
$ gcc -Wall ex2.c -o ex2

# Executing it
$ ./ex2
Compiler
Swapping C with r
Swapping o with e
Swapping m with l
Swapping p with i
relipmoC
```

The root of the activation tree represents the activation of the main() function. The main() activates the function string_reverse(), which in turn activates the function swap(), four times in the form of swap(0,7), swap(1,6), swap(2,5) and swap (3,4). Each of these swap() invocations call the my_print() function in the form of my_print('C', 'r'), my_print('o', 'e'), my_print('m', 'l'), my_print('p', 'i') respectively. The activation of the C library function 'printf()' in the activation tree is not shown to keep it simple. The flow of control in the program can be deduced by doing a depth first traversal of the activation tree. The traversal of the activation tree starts at the root of the tree. Each node is visited before its children are visited. The children are visited from left to right order. When all the children of a particular node have been visited, the function activation corresponding to the node is completed. The reader can verify that the depth first traversal of activation tree in Fig. 6.11 can recreate the program output.

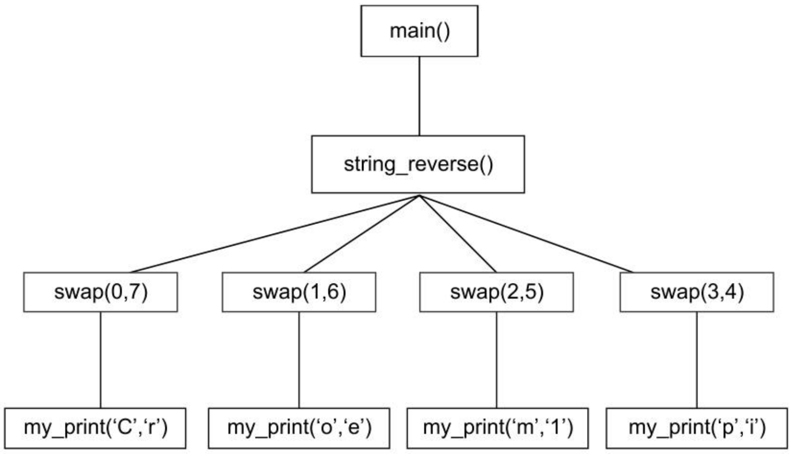


Fig. 6.11 *Activation tree*

A stack representing the flow of control, called as ***control stack*** or ***runtime stack*** is used to keep track of the activations that are in progress currently. When an activation of a function happens, a node corresponding to it is pushed on to the control stack. When the activation ends, the node is popped out of the control stack. Figure 6.12 shows an instance of control stack during the execution of string reversal program that we saw previously. It shows the control stack during the execution of my_print('m', 'l').

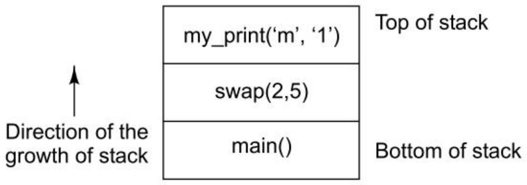


Fig. 6.12 *Control stack*

The control stack and the activation tree are related in an interesting way. When a node N() is at the top of the control stack, the other elements on the control stack are the ones that are along the path from N to the root. For example, the activation tree at the time of execution of my_print('m', 'l') is shown in Fig. 6.13. The path in the activation tree from my_print('m', 'l') to the root of the tree (main()) is shown in the dark line. This path given by my_print(m,l), swap(2,5), main() is the same as the control stack at that point, shown previously in Fig. 6.12. The activations of the functions connected using the dotted lines have executed to completion.

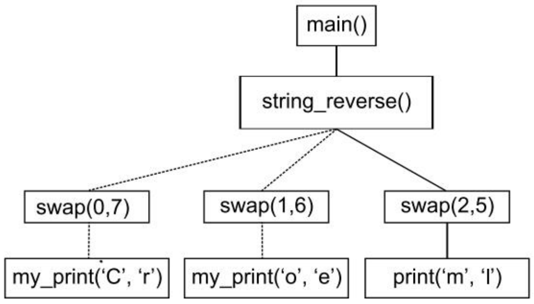


Fig. 6.13 *Activation tree*

Let's briefly touch upon the some other terminology that is used in some of the later sections.

When a variable, say 'v', is declared in a program, it needs to be mapped to a memory location 'L' at the runtime. This mapping is known as the ***binding*** of 'v' to 'L'. The location 'L' symbolises a set of memory locations equal to the size of the variable. The term ***data object*** is also to refer to the memory location a variable is mapped into.

The ***scope*** of a declaration is the region where the declaration applies. This is the region where the variables defined in the declaration can be used according to the rules of the programming language. For example, a variable declared within a function in C language would have a scope within the function. The variable cannot be used outside it.

The ***environment*** is a function that maps a variable name to a memory location. The ***state*** refers to the function that maps the memory location to the value stored in it. A data object can contain different values at different times.

We look at an example C program, which illustrates the idea of binding of variable to memory and also the state of the variable.

6.3.1.1 Example 3—A Program to Display the Binding and State of Variables

The dialog below shows a C program that prints the memory location to which a variable is bound and also its state. It can be observed from the output that the same variable can be bound to different memory locations at different times.

```
# Program to print the Bindings and states of Variables
$ cat -n ex3.c
     1  #include <stdio.h>
     2
     3  void func2();
     4  void func1();
     5
     6  int g=0;
     7
     8  int main()
     9  {
    10          func1();
    11          func2();
    12          return(0);
    13  }
    14
    15  void func1()
    16  {
    17          int i,j,k;
    18
    19          i=1;
    20          j=2;
    21          k=3;
    22
    23          printf("=============\n");
    24          printf("Variable 'i' is bound to %lx state=%d \n", (unsigned long)&i,i);
    25          printf("Variable 'j' is bound to %lx state=%d \n", (unsigned long)&j,j);
    26          printf("Variable 'k' is bound to %lx state=%d \n", (unsigned long)&k,k);
    27
    28          printf("Variable 'g' is bound to %lx state=%d \n", (unsigned long)&g,g);
    29          g++;
    30
    31  }
    32
    33  void func2()
    34  {
    35          int a,b,c;
    36
    37          a=10;
```

```
        b=20;
        c=30;

        printf("==============\n");
        printf("Variable 'a' is bound to %lx state=%d \n", (unsigned long)&a,a);
        printf("Variable 'b' is bound to %lx state=%d \n", (unsigned long)&b,b);
        printf("Variable 'c' is bound to %lx state=%d \n", (unsigned long)&c,c);

        printf("Variable 'g' is bound to %lx state=%d \n", (unsigned long)&g,g);

        func1();

}

# Compiling it
$ gcc -Wall ex3.c -o ex3

# Executing it
$ ./ex3
==============
Variable 'i' is bound to 73cb44 state=1
Variable 'j' is bound to 73cb40 state=2
Variable 'k' is bound to 73cb3c state=3
Variable 'g' is bound to 403010 state=0
==============
Variable 'a' is bound to 73cb44 state=10
Variable 'b' is bound to 73cb40 state=20
Variable 'c' is bound to 73cb3c state=30
Variable 'g' is bound to 403010 state=1
==============
Variable 'i' is bound to 73cb24 state=1
Variable 'j' is bound to 73cb20 state=2
Variable 'k' is bound to 73cb1c state=3
Variable 'g' is bound to 403010 state=1
```

6.3.2 Elements of a Run-time Environment

As stated earlier, Run-time environment describes the run-time settings of a program in execution. In this section, we introduce the important elements that constitute a run-time environment for a program. We study briefly about each one of those in this section. Later, we examine each of these elements in detail and get into some specifics about run-time environments provided by the compilers for FORTRAN, C and Pascal.

The important elements in a Run-time environment are:

(1) ***Memory Organisation:*** At the time of execution, the program requires memory for storing local variables, global variables, the code of the program, data structures for keeping track of the activations, and so on. One of the important characteristics of a run-time environment is the way memory is organised during execution. The features of the source language determine the way run-time memory is organised. For example, FORTRAN77 specifications did not support pointers or usage of any dynamic memory. It did not support recursion in procedures. This is in contrast to a source language like C where, there is support for pointers and use of dynamic memory in the form of malloc () and free () routines. C language allows recursion in its functions. The memory organisation in FORTRAN77 run-time environment would be very different from C run-time

environment given the contrast in the language features. The way memory is organised is an important aspect of a runtime environment.

(2) ***Activation records:*** Typically, source languages support procedures or functions for making the programs modular. Procedure activation is managed by having a contiguous block of memory called the ***activation record***. The activation record contains among other things, the memory for all the local variables of the procedure. A single activation record which is common across any number of activations can be created statically. The activation record can also be constructed dynamically, one for each *activation*. Depending on how the activation record is created, the target code has to be generated accordingly to access the local variables that are part of it. In block-structured languages like C and Pascal, the activation record which houses memory for the local variables would also include the space for the variables declared in different blocks of the procedure. The scoping rules of the source language dictate how the target code needs to be generated in order to access the variables in the block. The content of activation record, the code generated to access the variables given the scoping rules of the language, the method used for creating activation record, and the place where activation records are stored are all important aspects of the run-time environment.

(3) ***Procedure calling and return sequences:*** When a procedure is activated, there are certain sequence of operations like evaluating the function arguments, placing it in a mutually agreed upon location between the caller and callee, transferring the control to the called procedure, and so on. This sequence of operations that are carried out during the process of calling a procedure is known as the ***calling sequence***. Similarly, when an activated procedure completes execution, there would be a sequence of actions to be performed like fetching the return value from a mutually agreed upon location between the caller and callee, transferring the control back to the caller, etc. This sequence of actions to be carried out, when a procedure returns after completing its execution is known as ***return sequence.*** The calling and return sequences, the division of responsibility between caller and the callee in these sequences are vital aspects of a runtime environment. They differ from one source language to another and in a few rare cases from compiler to compiler for the same language.

(4) ***Parameter passing:*** When a function is called, it can be passed one or more parameters. The called function might modify the value of the parameter. Some of the source languages like PASCAL and C++ specify rules in which context the modified value should reflect in the caller and in which it should not. In some of the source languages like FORTRAN77, the modification of parameter in a function always reflects in the caller. There exist several mechanisms by which parameters can be passed to functions. The target code generator should take into account the type of the parameter-passing mechanism used in the context and generate code accordingly. An important characteristic of a given run-time environment is the support provided for different parameter-passing mechanisms.

In the next few sections, we study about each of these elements of a run-time environment in detail. The discussion is supplemented by taking examples of how things are structured in run-time environments of common programming languages like C, FORTRAN77, Pascal, ADA, and so on.

6.3.3 Memory Organisation

The operating system provides a block of memory for executing a compiled program. The block of memory is segregated into logical areas for ease of execution. The layout of a program in memory ready for execution illustrating the various logical divisions in memory is shown in Fig. 6.14.

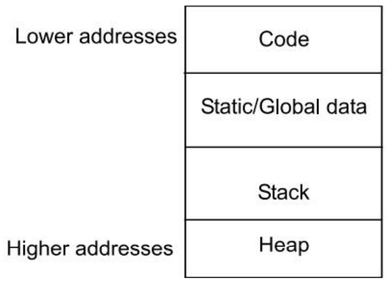


Fig. 6.14 *Memory layout*

The program instructions consisting of the code for each of the procedures is stored in the code region (also known as the program memory). The size of the code for each of the procedures is known at the time of compilation, so the total size of this area is fixed at the compile time for a given program.

The static/global data region provides the memory for storage of all the global variables declared in the program. It also provides storage for all the static objects (which do not change during the execution) like string literals, large constants, etc. used in the program. The amount of memory needed for this section is also known at the time of compilation similar to the program region.

The stack and heap area are used for allocation of dynamic memory required by the program. The dynamic memory is the memory that can be allocated and freed during the execution of program as opposed to the static one, which stays allocated throughout the life of the program.

The stack area is used for allocating memory to data that follow last-in-first-out (LIFO) order of allocation to de-allocation. In other words, the stack houses data structures in which last chunk to be allocated would be the first one to be freed.

The heap region provides for other dynamic memory needs in a program that do not follow the last-in-first-out order of allocation to de-allocation.

To get an idea of what kind of data goes into the stack and the heap in a run-time environment, let's take the example of C program runtime environment. In a C program, when memory is requested by the program using dynamic memory management routines like say malloc(), calloc(), etc., it is provided from the heap. After using the obtained memory, the program can free the memory using the free() routine. The heap then reclaims the storage for the freed memory, so that it can be allocated to future memory allocation requests. The stack holds all the local variables defined within a function during its activation. The memory for local variables is de-allocated automatically after the activation of the function is complete. There is no explicit 'free' made in the program for the de-allocations on the stack.

The heap and stack grow and diminish during the lifetime of a program. Some of the compilers might have a run-time environment where stack starts at lower address and heap starts at higher address. There are others where the heap starts at lower address and stack starts at higher address. Regardless of where they are positioned, they can only be known at the run-time and keep growing and diminishing depending on the program activity.

In FORTRAN77 run-time environment all the data variables (both local and Global) are bound to the static memory region. Every variable is associated with a fixed address computed at the compile time that can be used to access it throughout the life of the program. The FORTRAN77 specifications did not support dynamic memory allocation and hence its run-time environment did not need a heap or stack.

6.3.3.1 Example 4—Memory Organisation in C Run-time Environment The example shown in the dialog below illustrates the memory organisation in a C run-time environment. The program displays the address of the variables stored in static area, stack area and also the heap area. The address of the functions stored in code area is also displayed. From the output, one can understand the partitioning of the runtime memory into specific areas.

A utility called 'objdump' also helps displaying the range of addresses of specific sections in a given executable. In the dialog below, we see 'objdump' deciphering the executable and showing the range of addresses in the code area (called text) and static area (given together by rdata, bss and idata). This establishes the fact that the code area and static area are determined at the compile time, and that information is stored in the object file. The heap and stack information can be determined at the runtime only. The reader is advised to read the manual pages of objdump for more details.

```
# Program to display addresses of variables
# Compiling it
$ gcc -Wall ex4.c -o ex4

# Executing it
$ ./ex4
========== CODE AREA ============
Start address of the main() in code area=401094
Start address of the func1() in code area=401050
Start address of the func2() in code area=4011c4
========== STATIC AREA ============
Address of the Global Variable g1 in Static area=403040
Address of the Global Variable g2 in Static area=403030
========== STACK AREA ============
Address of Local Variable a=73cb34
Address of Local Variable b=73cb30
Address of Local Variable e=73cb14
Address of Local Variable f=73cb10
========== HEAP AREA ============
Address allocated on the heap=b90518
Address allocated on the heap=b90528
Address allocated on the heap=b90538

# Using the objdump utility to get the addresses
$ objdump -h ex4.exe

ex4.exe: file format pei-i386

Sections:
Idx Name       Size         VMA       LMA       File off  Algn
 0  .text      00000520     00401000  00401000  00000400  2**4
               CONTENTS,    ALLOC,  LOAD,  READONLY,  CODE
 1 .rdata      00000250     00402000  00402000  00000a00  2**4
               CONTENTS,    ALLOC,  LOAD,  READONLY,  DATA
 2 .bss        00000070     00403000  00403000  00000000  2**4
               ALLOC
 3 .idata      00000188     00404000  00404000  00000e00  2**2
                            CONTENTS,  ALLOC,  LOAD,  DATA
```

6.3.4 Activation Records

The execution of a procedure is managed by maintaining a contiguous block of storage at the run-time called as ***activation record***. An activation of a procedure is associated with an activation record. For example, in the activation tree shown in Fig. 6.11, there are 4 instances of swap() being activated one after the other. In the C program, When swap(0,7) is activated, an activation record corresponding to it is created for aiding its execution and then destroyed at the end of execution. Similarly, when swap(1,6) is activated in the C program, another activation record corresponding to it is created for assisting its execution, and so on.

An activation record holds space for arguments to the procedure, local variables, compiler-generated temporaries, the return value of the function and some other fields that help the activation of a procedure. An activation record is illustrated in Fig. 6.15. The format of the activation record need not exactly adhere to the one shown in Fig. 6.15 for all the languages in different run-time environments. The exact content

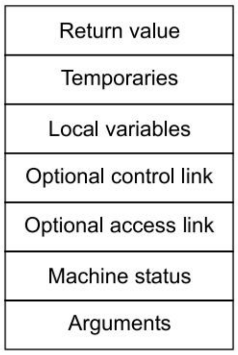


Fig. 6.15 *Activation record*

of the activation record depends on factors like the target architecture, the source language compiled, and so on. However, the fields of the activation record illustrated in Fig. 6.15 are found commonly in most of the activation records.

When a procedure is activated, storage space is required for local variables, arguments to the function, compiler-generated temporaries and the return value of the function. The activation record serves as a placeholder for all of these elements.

From a program control point of view, when a procedure is called, the control is transferred to the callee and after its execution is complete, the control is given to the next executable instruction after the function call in the caller. The return address in the caller, where the control should return after the callee completes execution is saved before the execution of the callee. It would be used to transfer the control back to the caller after the execution of callee. On most of the target architectures, a register called the program counter (PC) holds the address of the next instruction to be executed. The PC should be saved before transferring the control to the callee procedure and restored after its completion in order to achieve proper flow of control. The registers of the processor would be used freely by the callee for executing its instructions. If the registers are not saved before the execution of callee, and restored after its completion they would become corrupted due to the callee procedure execution. If the registers get corrupted, the caller procedure would not execute as expected after it returns from the callee. Hence when a procedure is called, the ***machine status*** in the form of contents of the registers, return address and the program counter should be saved before the call and restored after the execution of the called procedure is complete. The machine status is saved in the activation record before the call to a procedure and restored after the execution of the procedure.

In source languages like Pascal, procedures can be nested. Consider a Pascal program having nested procedures P1 and P2 as shown in the Listing 6.8. The procedure ‘P2’ is nested within ‘P1’. The variable ‘c’ used in the nested procedure P2 is defined in P1. At the place where c is used, i.e. line 12, it should be resolved to the declaration in P1. In order to implement such scope rules that allow resolving the names unambiguously to a declaration in the case of nested procedures at the compile time, we have an optional field called as ***access link*** in the activation record. The access link typically points to the activation record of the defining environment. We learn about the exact usage of access link field in implementing the scope rule for nesting of procedures later in the chapter. This optional field is not required in the activation records for source languages like C where no nesting of procedures is allowed.

```
 1  PROGRAM sample(input,output);
 2  VAR a,b: integer;
 3
 4          PROCEDURE P1();
 5          VAR a,b : integer;
 6          VAR c : integer;
 7                  PROCEDURE P2();
 8                  VAR a,b : integer;
 9                  BEGIN
10                              a:=10; (* This should resolve to ‘a’ defined in P2 *)
11                              b:=20; (* This should resolve to ‘b’ defined in P2 *)
12                              c:=25; (* This should resolve to ‘c’ defined in P1 *)
13                  END;
14          BEGIN
```

```
            c :=40;
            writeln('Value of c is ',c); (* c is 40 here *)
            P2();
            writeln('Value of c is ',c); (* c is 25 here *)
      END;
BEGIN
      P1();
END.


```

Listing 6.8 A Pascal program with nested procedures

Programming languages like LISP and APL allow variables to be bound to a storage depending on the current activations. In these cases the variable can be resolved to the appropriate declaration only at the run-time depending on the current activations. In order to implement such dynamic scoping, it is necessary to keep track of the chain of the current activations. The optional ***control link*** in an activation record helps in maintaining a track of the current activations and implementing dynamic scope. Following the control link of the current activation record, we can make a chain of all the functions that are currently active. This helps in implementing the dynamic scope.

The activation record contains a field for storing the return value of a function. The callee stores the return value in this field before returning the control to the caller. The caller copies it from this field into the appropriate variable as defined in the source program. In practice, many of the compilers, have the return value and the arguments passed in registers, whenever feasible rather than having them as a part of activation record. The register access is faster than memory access and hence passing the return values and arguments in registers is more efficient.

Activation records are allocated space in the stack Area in C run-time environment. The Old FORTRAN77 compilers used the static area for housing the activation records. The run-time environments for functional languages like LISP allocate space for activation records on the heap.

6.3.4.1 Activation Record in C Run-time Environment In C runtime environment, the activation records are allocated storage space on the stack. When a procedure is called, a new activation record is pushed on to the stack. When the procedure is complete, the activation record is popped-out of the stack. The top of the stack is usually pointed to by a register called SP (stack pointer). An activation record can be allocated by moving SP with an amount equal to the size of activation record. The activation record is de-allocated by moving the SP back by an amount equal to the size of activation record. For example, consider the activation of a function 'my_func()' having an activation record of size, say, 40 bytes. The SP is moved (decremented in this case) by 40 bytes to allocate an activation record for my_func(). The SP is moved back (incremented by 40) to de-allocate the activation record for my_func() after the execution of my_func() is complete. Figure 6.16 shows the run-time stack, before, during and after the activation of my_func().

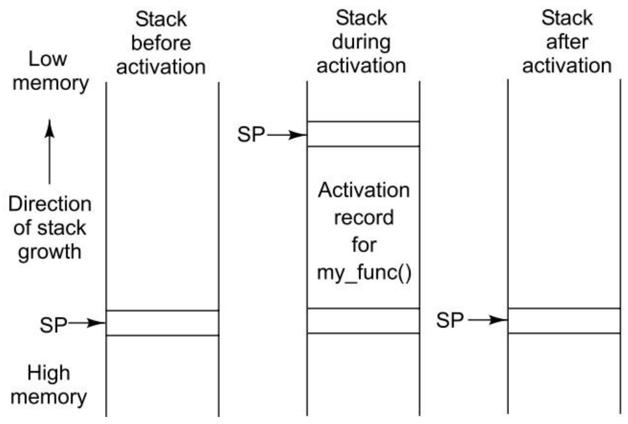


Fig. 6.16 *Allocating and de-allocating space for activation records*

Another register known as frame pointer (FP) or base pointer (BP) stores a pointer to the current activation record. The BP points to a location in the current activation record that is used as a base for accessing the local variables, arguments to the function, temporaries and other elements in the activation record. Figure 6.17 shows the stack layout during the activation of function ‘my_func’ and how BP can be used for accessing arguments and local variables.

The local variables in the currently active function are accessed using negative offsets from the location where BP points. In the above example, during the time that function ‘my_func’ is active, the local variable ‘ret’ is accessed as –4(%BP), ‘c’ as –8(%BP), ‘d’ as –12(%BP). Consider the initialisation of the local variables ‘c’ and ‘d’ in the function ‘my_func’ of Fig. 6.17. The code generator produces the following x86 assembly code for initialisation using the relative offsets of the variables ‘c’ and ‘d’ from the BP.

```
int my_func(int a,int b)
{
    int ret;
    int c_=10,d=20;

    c=a*a;
    d=b*b;
    ret=c+d+2 *a *b;

    return(ret);
}
```

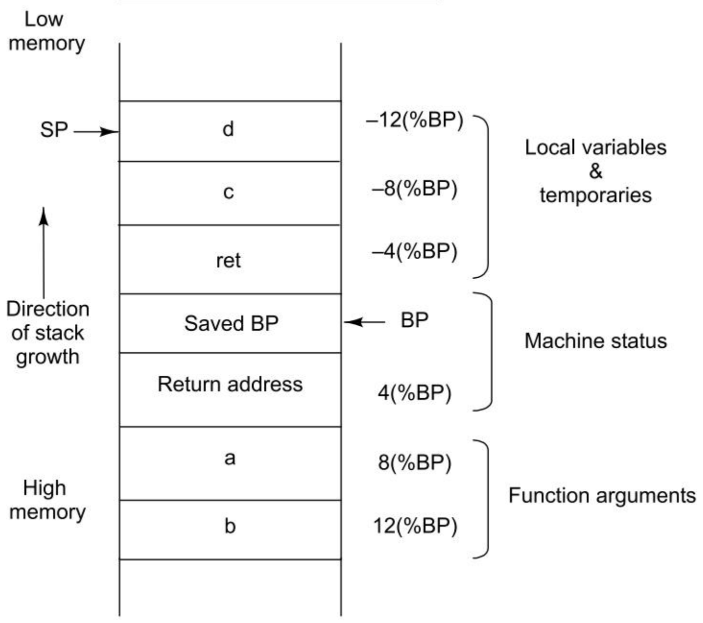


Fig. 6.17 *Stack during the activation of a function*

```
movl  $10,-8(%ebp)  # Variable ‘c’ is at offset –8 from the BP throughout the function
movl  $20,-12(%ebp) # Variable ‘d’ is at offset –12 from the BP throughout the function
```

The parameters in the currently active function are accessed using positive offsets from the location where BP points. In the above example, during the time that function ‘my_func’ is active, the parameter ‘a’ is accessed using 8(%ebp) and parameter ‘b’ using 12(%ebp). Consider the computation for the line ‘c=a*a;’ in the function ‘my_func’ of Fig. 6.17. The code generator produces the following x86 assembly code for the computation using the relative offsets of the parameters ‘a’ and ‘b’ from the BP.

```
movl 8(%ebp),%eax  # Accessing the parameter ‘a’
imull 8(%ebp),%eax # Accessing the parameter ‘a’ again and multiplying it
movl %eax,-8(%ebp) # Moving the result to ‘c’
```

The layout of the activation record in terms of the offsets of the local variables and arguments from the location that BP points, remains exactly the same any time the activation of the function happens.

For example, the memory for the variable 'ret' would always be at an offset –4 from the BP, any time the function 'my_func' is activated irrespective of the caller.

Till the time that the program control is within the function my_func, the offsets from BP for the local variables, arguments to the function, compiler generated temporaries, etc. remain unchanged.

The creation of activation record on the stack allows for having multiple instances of activation records, each one corresponding to an activation of a procedure. For example, consider a recursive routine to evaluate the factorial of a given number shown in Fig. 6.18.

```
1 #include <stdio.h>
2
3 int main()
4 {
5    int x;
6    x=factorial(5);
7    printf("%d\n",x);
8 }
9
10 int factorial(int n)
11 {
12    if(n==1){
13     return(1);
14    }
15    return(n*factorial (n-1));
16 }
17
```

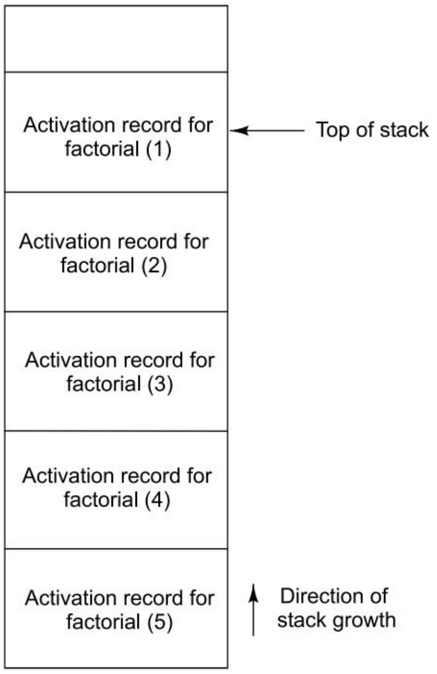

Activation record for factorial (1)
Top of stack
Activation record for factorial (2)
Activation record for factorial (3)
Activation record for factorial (4)
Activation record for factorial (5)
Direction of stack growth

Fig. 6.18 *Stack showing multiple activation records*

The first activation from main() as factorial(5) creates an activation record for factorial(5) on the stack. The next activation comes due to recursive call made at line number 15 with factorial(4). The next activation is due to the call again from line 15, this time as factorial(3), and so on. The activation records stack up on the run-time stack as seen in Fig. 6.18. This facility of having an instance of activation record in the memory, one for each of the activations helps in supporting recursion.

We will now look at an example program, which displays the content of activation record in C run-time environment provided by the C compiler(cc1) of the gcc—The GNU compiler collection. This activation record has the following format.

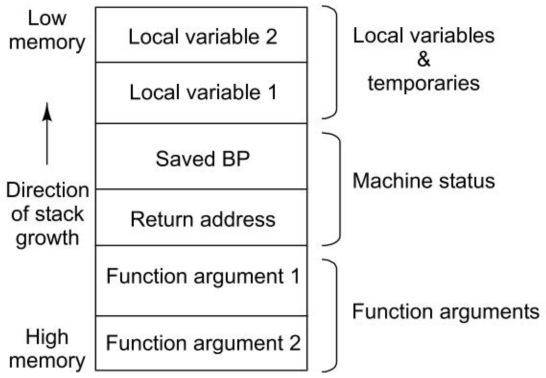


Fig. 6.19 *Activation record for a C program (gcc)*

The dialog below shows the compilation and execution of the example C program.

```
# Program to print the Activation record in C
# Compiling it
$ gcc -Wall ex5.c -o ex5

# Executing it
$ ./ex5
THE ACTIVATION RECORD OF func1
======================================
Start of Activation Record  [addr=0073cb44]
Parameter2       : 22222222 [addr=0073cb44]
Parameter1       : 11111111 [addr=0073cb40]
Return Address   : 0040109d [addr=0073cb3c]
Saved BP         : 0073cb68 [addr=0073cb38]
Local Variable1  : 33333333 [addr=0073cb34]
Local Variable2  : 44444444 [addr=0073cb30]
```

6.3.4.2 Activation Record in FORTRAN77 Runtime Environment In FORTRAN77 run-time environment, for each procedure in the program, an activation record is statically allocated at the compile time. Consider a FORTRAN77 program that has 3 procedures P1, P2, P3. The FORTRAN77 environment is a completely static one, where all the activation records are created at the compile time and stored in the static memory area. Figure 6.20 shows the memory organisation of the program in such a run-time environment.

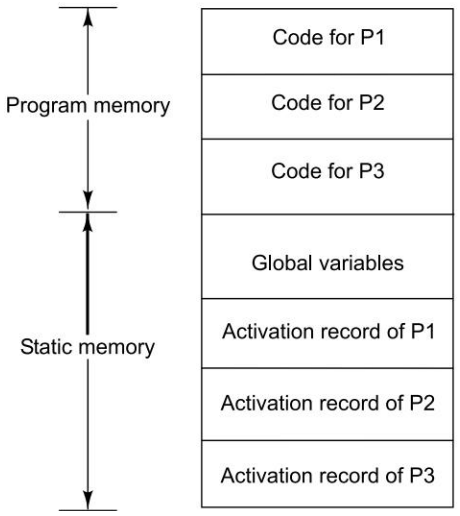


Fig. 6.20 *Activation records in FORTRAN77 run-time environment*

The local variables in procedure P1 would be part of its activation record and would be bound at the compile time. In a similar way, the local variables of P2 and P3 would be part of their respective activation records and bound at the compile time. Any procedure has only one activation-record and multiple activations would have to use the same activation record. Hence, this type of run-time environment cannot support recursion in procedures. The fixed activation record means that any local variable in a function is bound to a fixed memory location throughout the life of the program. The side-effect of this is that the values of local variables can be retained across the activations of a procedure.

In FORTRAN77 environment, the activation records of all the procedures are stored in the static memory throughout the lifetime of the program. This wastes memory, since the activation record for a procedure gets used only when it is activated.

6.3.4.3 Non-Local Access We have seen how the activation record helps us in managing the procedure activation and handling access to local variables. We will now look at how activation records can also help us access variables defined in the enclosing scope.

Programming languages commonly offer features for accessing variables that are not only declared locally, but also declared in an enclosing scope. For example, C language allows for having ‘blocks’ of code, where variables that are declared in an enclosing block can also be accessed. PASCAL allows for procedures to be nested. In the nested procedure, We can access variables that are declared in the enclosing procedure. These are two good examples of accessing ***non-local*** variables defined in an enclosed scope of a program. We use the term non-local to essentially mean that they are not defined in the Local scope. It is pertinent at this point to make the distinction between a ‘non-local’ variable and a global variable clear. A global variable is in the outermost scope and is treated differently by having storage allocated in the static memory and any access to it would be resolved to that static memory location at the compile time. A non-local variable is simply a variable that is neither declared in the current scope nor the outermost scope, it is declared in-between these.

In both of the above cases of non-local access, (i.e. in the case of blocks or nested procedures) the scoping rules of the respective languages C and PASCAL specify that the reference to the variable be resolved to the nearest enclosed declaration made in the program text. This type of scoping, where a reference to the variable can be resolved to the appropriate declaration by examining the program text regardless of the current activations is known as ***lexical*** or ***static scope*** rule. This is in contrast with ***dynamic scope*** rule used in source languages like LISP, SNOBOL and APL, in which a variable can be resolved into the correct declaration only during the execution by considering the current activations. The interpreted language PERL supports both lexical and dynamic scoping.

In order to make the distinction between lexical scoping and dynamic scoping clear, let’s consider the 2 PERL programs seen in the dialog of Fig. 6.21. The first program ‘dynamic.pl’ shows the dynamic scoping at work. The execution of dynamic.pl shows that the function ‘g’ returns different values depending on where it is in invoked. When it is invoked from within the function ‘f’, the value returned is that of the variable ‘y’ defined within ‘f’. The variable ‘y’ is specifically declared within function ‘f’ as a dynamically scoped variable by the use of keyword ‘local’. When ‘g’ is invoked from the main, it returns the global variable ‘y’, since that is active in the scope. The second program ‘static.pl’ shows static scoping at work. In this program, the variable ‘y’ declared within the function ‘f’ is lexically scoped. Any access of the variable ‘y’ within the function ‘f’ would refer to the declaration in ‘f’. However, all the other routines like say ‘g’ continue to refer to the global declaration of ‘y’. Hence when ‘static.pl’ is executed, the value returned by the function ‘g’ is consistent regardless of called site.

```
# dynamic.pl
$ cat -n dynamic.pl
     #!/usr/bin/perl


     $y=101 ;

     sub f () {

             local $y ;        # Dynamically scoped variable

             $y = 10 ;

             return g();


     }
```

```
    16
    17  sub g () {
    18          return $y ;
    19  }
    20
    21
    22  print “y at the start = $y \n” ;
    23
    24  print “y when g is called directly=”,g(),”\n” ;
    25  print “y when g is called indirectly=”,f(),”\n” ;
    26
    27

# Executing it
$ ./dynamic.pl
y at the start = 101
y when g is called directly=101
y when g is called indirectly=10

# static.pl
$ cat -n static.pl
     1  #!/usr/bin/perl
     2
     3
     4  $y=101 ;
     5
     6  sub f () {
     7
     8          my $y ;   # Lexically scoped variable
     9
    10          $y = 10 ; # This refers to the ‘y’ declared above
    11
    12          return g(); # g continues to access the global ‘y’
    13
    14
    15  }
    16
    17  sub g () {
    18          return $y ;
    19  }
    20
    21
    22  print “y at the start = $y \n” ;
    23
    24  print “y when g is called directly=”,g(),”\n” ;
    25  print “y when g is called indirectly=”,f(),”\n” ;
    26
    27

# Executing it
$ ./static.pl
y at the start = 101
y when g is called directly=101
y when g is called indirectly=101
```

Fig. 6.21 *Lexical and dynamic scoping*

In this chapter, we study about resolving the access to the variable references in static scope only. The resolving of variable references in dynamic scope is not taken up.

We now discuss about how the activation records help us implement lexical scoping and resolve access to the non-local variables that are declared in the enclosed block in a block-structured language like C. Later we learn about how activation record can also be used to resolve accesses to the non-local variables that are declared in an enclosing procedure. This would apply to languages like PASCAL that allow nesting of procedures.

Non-local access in the context of blocks A block is a set of statements containing its own local declarations. A block is identified by a pair of delimiters. In C language, the delimiters for a block are the curly braces '{ and }'. In Pascal, the delimiters for a block are 'begin' and 'end'. The delimiters ensure that blocks do not overlap each other. For example, there cannot be a situation where a block B1 begins, followed by the beginning of block B2 and then block B1 ends before B2. Blocks can only be nested like say block B1 begins, followed by beginning of B2 , then B2 ends, before the end of B1. In Listing 6.9, there are 4 blocks of code in the main() function. The block B0 spans from line 4 through 43, block B1 from line 10 through 36, B2 from line 14 through 22, B4 from line 24 through 32. Each of these blocks has its own declarations for the variables *v*1 and *v*2.

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5       /* Block B0 Begins */
 6
 7       int v1=10,v2=20;
 8       int v3=5;
 9
10       {
11            /* Block B1 Begins */
12            int v1=20,v2=30;
13
14            {
15                 /* Block B2 Begins */
16                 int v1=40,v2=50;
17
18                 v3=90;
19                 printf("In Block B2 v1=%d v2=%d v3=%d \n",v1,v2,v3);
20
21                 /* Block B2 ends */
22            }
23
24            {
25                 /* Block B3 Begins */
26                 int v1=60,v2=70;
27
28                 v3=80;
29                 printf("In Block B3 v1=%d v2=%d v3=%d \n",v1,v2,v3);
30
31                 /* Block B3 ends */
32            }
33            printf("In Block B1 v1=%d v2=%d v3=%d \n",v1,v2,v3);
34
35            /* Block B1 ends */
36        }
37
38        printf("In Block B0 v1=%d v2=%d v3=%d \n",v1,v2,v3);
39
40        return(0);
41
42        /* Block B0 ends */
43   }
44
```

Listing 6.9 *blocks.c*

In the example shown in Listing 6.9, the access of the variables *v*1 and *v*2 in the printf statement shown in line 19 (block B2) would be resolved to the declaration in line 16 (block B2). This is an evidence of what is known as 'the most closely nested rule' in resolving the scope. In simple terms, what this means is that any access to non-local variables should be resolved to the block that is most closely nested from the point of access. The declaration of *v*1 and *v*2 on line 16 is in the most closely nested block B2. This declaration is considered ahead of the one at line 12, which is in the enclosing block B1. The most closely nested rule is used again to determine that the access for variables *v*1 and *v*2 in line 29 (B3), which is resolved to declaration in line 26 (B3) ahead of the declaration at line 12 in the enclosing block B1.

The most closely nested rule for resolving the scope of declarations in a block-structured language can be stated as follows:

1. The scope of a declaration for a variable *v*1 in a block B pervades throughout the entire block B.
2. If a variable *v*1 is not declared in a block B, but used in B, then it is resolved to the declaration in a block B1, which encloses B. In case there are two enclosing blocks B1 and B2 that contain declarations for *v*1, the innermost block is taken for consideration.

The block structure and the most closely nested rule can be implemented by having a separate symbol table created for all the variables declared in a block. When a variable is used in an execution statement of a block, the symbol table for the most deeply nested block is consulted first, followed by the enclosing block, and so on, until the symbol table for global scope.

At the run-time, the most closely nested rule in a block is supported by allocating memory on the stack for the variables in a block. There are a couple of schemes for allocating memory on the stack with respect to the block variables. Let us take the example of the C program—blocks.c, given in Listing 6.9 and understand the two schemes for allocation of memory to the block variables.

In the first scheme, when a block is entered, the space for all the variables declared in it is allocated on the stack. When the control leaves the block, the memory allocated for the same variables is discarded. Figure 6.22 shows the snapshot of the stack when the control is in block B1, when this scheme is used.

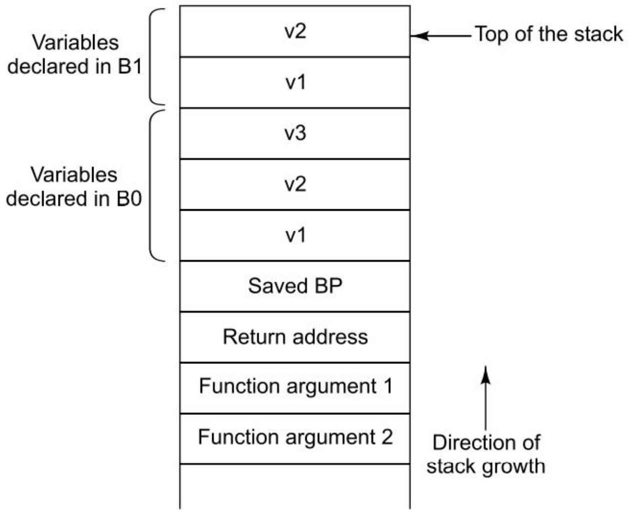


Fig. 6.22 *Snapshot of stack when the control is in block B1*

We can see that the variables declared in block B1 are present on the stack and so are the ones declared in B0. This allows for access of variables declared in the enclosing scope like *v*3, which are at a known offset on the stack. The assignment to *v*3 in the block B2 would be typically done by generating x86 assembly instructions like the one show below, given that the memory for the variable is at a known fixed offset on the stack.

```
movl $90, -12(%ebp)
```

We can see from Fig. 6.22 that the usage of stack for handling of blocks is very similar to the handling of declarations in procedures seen earlier. The only difference being that the saving of machine status, return address, etc. on the stack is not performed in the case of blocks. A block can be viewed as a parameter-less procedure called from the point where the block begins and returning at the point of ending of the block.

In the second scheme, the memory needed for all the blocks in the entire procedure is allocated at the start of the function as a part of the activation record. The memory necessary for all the blocks in a procedure is computed by taking the lifetimes of the blocks into consideration. For example, in the procedure shown in the Listing 6.9, the blocks B2 and B3 have mutually exclusive lifetimes, i.e. either the block B2 is alive or block B3 is alive at any given point of time. The memory required for the blocks in 'my_func()' is computed by taking the sum of the memory required for blocks B0, B1 and the greater of B2 and B3 (since only one of them is active at any point). For a procedure having multiple control paths, the memory required for a procedure is determined by traversing all the control paths in it and computing the memory required for them. The cumulative memory required by all the blocks is computed and allocated at the start of the function. In this scheme, the memory for any variable declared in a block can be resolved at the compile time by using the tuple (variable name, block number), since the variable name alone does not suffice. The resolving of the variable to the memory happens at the compile time and hence it is possible to generate instruction for accesing the variables in the block.

In both of the schemes mentioned above, the non-local variable defined in one of the enclosing blocks is resolved into an address on the stack at a known offset. The reader is advised to compile the program shown in Listing 6.9 and observe the x86 assembly output to verify the above-mentioned principles of resolving the address of the non-local variables.

Non-local access in the context of Nested Procedures Source Languages like Pascal allow procedures to be nested. These nested 'local procedures' can be called from the procedure defining it or from one of its peers. Consider a sample Pascal program with local procedures shown in Listing 6.10. The local procedure P2 is nested inside P1. The procedure P2 is called in the action statements of the procedure P1 (Line 31) and in a peer procedure, e.g. Peer_Of_P2 (Line 16) as seen in the listing. The scoping rules in Pascal allow the variables defined in an enclosed procedure to be used in the local procedures. For example, the variable '*v*1' is defined in the procedure P1 at Line 5 and is used in the procedure P2 at line 11.

```
 1  PROGRAM sample(input,output);
 2  VAR v0 : integer;
 3
 4          PROCEDURE P1(param1:integer);
 5          VAR v1 : integer;
 6
 7               PROCEDURE P2();
 8               VAR v2: integer;
 9               BEGIN
10                    v2:=20;
```

```
                    v1:=25; (* 'v1' defined in P1 *)
               END;

               PROCEDURE Peer_Of_P2();
               BEGIN
                    P2();
               END;


               PROCEDURE Another_Peer_Of_P2();
                    PROCEDURE P3();
                    BEGIN
                         P2();
                    END;
               BEGIN
               END;

          BEGIN
               if (param1 > 100 )
               then
                    P2()
               else if( param1 > 50 )
               then
                    Peer_Of_P2()
               else
                    Another_Peer_Of_P2() ;

          END;

BEGIN
          write('Give Input :');
          readln(v0);
          P1(v0);
END.
```

Listing 6.10 *nested_proc.pas*

At the time of execution of the above program, the memory for variable 'v1' is allocated on the stack corresponding to the procedure P1, since it is defined locally in P1. Being a local of P1, the memory for variable 'v1' would be a part of its activation record. Figure 6.23 shows the stack layout of the program during the execution of procedure P2(), when invoked from P1(). The relevant variables 'v1' and 'v2' in the activation records of P1() and P2() respectively are shown in Fig. 6.23. The other fields in the activation records are not shown in Fig. 6.23 for clarity purposes.

Figure 6.24 shows the stack layout during the execution of procedure P2, when invoked from Peer_Of_P2(), which in turn is called from P1(), when the given user input is between 50 and 100.

We can see from the two stack layouts that the variable 'v1' would not be at a fixed known offset on the stack from the activation record of P2. This poses a challenge for resolving the variable 'v1' to the appropriate address on the stack at the compile time. The local variables of P2 like 'v2' can be accessed by indexing from the BP as say –4(%ebp) , –8(%ebp), etc. but 'v1' cannot be accessed in a similar way, because the offset from the BP is different depending on the caller of P2 as seen in stack layout 1 and 2.

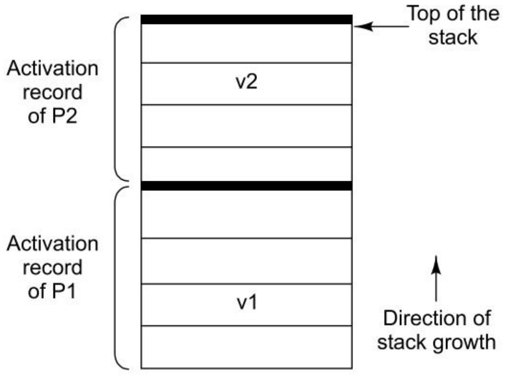


Fig. 6.23 *Stack layout—1*

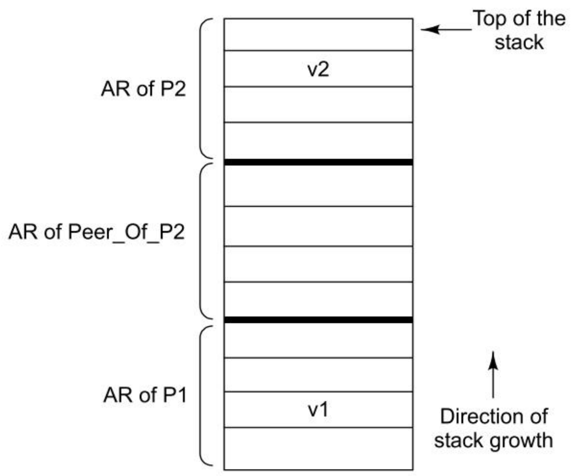


Fig. 6.24 *Stack layout—2*

The solution to this issue is to have a link to the enclosing procedure's activation record (in this case, P1) in the current activation record and use that to index to the exact address corresponding to the variable on the stack. We are taking advantage of the fact that the position of 'v1' within the activation record of P1 is fixed and known at the time of compilation. This link to the enclosing procedure's activation record is called as the ***static link*** as opposed to the dynamic link which points to the activation record of the caller. In the above example, the activation record of P2 would contain a static link field that points to the activation record of the enclosing procedure P1. This is irrespective of the caller of P2(), which can be either P1, Peer_of_P2() or Another_ Peer_Of_P2(). Once we have the pointer to the activation record of P1, the variable 'v1' can be accessed using a specific offset since it is a local of P1. Figure 6.25 shows the use of static link to access the memory for the variable 'v1', in the scenario corresponding to stack layout-2.

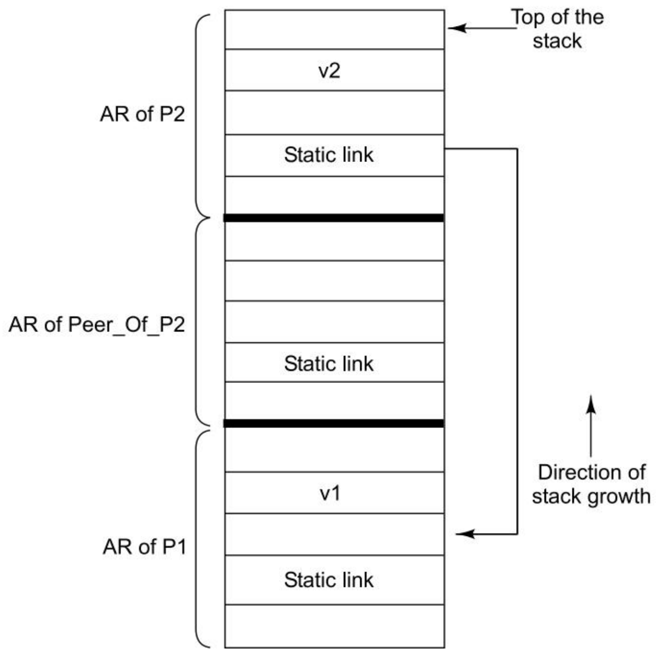


Fig. 6.25 *Use of static link*

In order to implement the static link and support non-local variable access in nested procedures, a concept called as lexical level is used. A ***lexical level*** in a pascal program corresponds to the nesting level in the source program. The main program is at lexical level 0. All the global variables declared in the main program are at the lexical level 0. For example, the variable ‘v0’ declared in line 2 of the Listing 6.10 is at lexical level 0. All the procedures declared at one level from the main program are at lexical level 1. In the Listing 6.10, the procedure P1 is at lexical level 1, the procedures P2, peer_of_P2() and another_Peer_Of_P2() are all at lexical level 2 and P3 is at lexical level 3. In the execution statements of the program, variables defined in the current level or lesser than the current level can be accessed. We can see in the Listing 6.10, the procedure P2 (lexical level 2) accesses variable ‘v1’ defined in the enclosing procedure P1 at lexical level 1, P2 also accesses variables defined in its own level ‘v2’.

Using the idea of lexical levels in the nested procedures, the procedure calling conventions in PASCAL language can be stated as follows.

(a) A procedure at lexical level ‘L’ can call a child procedure i.e. the one immediately nested in it, at lexical level (L + 1). In Listing 6.10, we can see that the main program, which is at level 0, invokes the procedure P1 (at line 43), which is at lexical level L.

(b) A procedure can call one of its ancestors. The ancestor is a procedure, which is a parent (immediately enclosing procedure), or a grandparent or one of the peers of a parent/grandparent. In other words, a procedure at lexical level ‘L’ can call one of its ancestors at lexical level less than ‘L’. In Listing 6.10, we see the procedure P3() at lexical level 3, calling Peer_of_P2() (at line 23), which is at lexical level 2—a peer of its parent P2.

(c) A procedure can call any one of its peers at the same lexical level. In Listing 6.10, we also see the procedure Peer_of_P2() at lexical level 2 calling its peer procedure P2 also at lexical level 2 in line 16.

The target code generator produces code that uses the static link field in an activation record and enables non-local variable access in nested procedures in all of the 3 cases (a), (b) and (c) mentioned above. The scheme used by the target code generator can be broadly described as follows.

(1) The caller passes the pointer to the activation record of the defining environment to be used as the static link in the caller to access the non-local variable. The pointer to the activation record of the defining environment is typically pushed on to the stack by the caller before making a function call and is popped out after the call is complete, similar to the function arguments.

(2) The callee uses the static link and accesses the memory for the non-local variable defined in the enclosing procedure using its offset.

Let’s take the example shown in Listing 6.10, dissect its x86 assembly language output generated by the PASCAL compiler and understand how the above-mentioned steps are accomplished.

The process of compilation and generation of assembly language output for ‘nested_proc.pas’ shown in Listing 6.10 by using gpc—A Pascal compiler, is given in the following dialog.

```
# Program illustrating nested procedures
$ gpc -Wall –save-temps nested_proc.pas -o nested_proc

# The x86 assembly language output for nested_proc.pas
$ cat -n nested_proc.s
     1          .file "nested_proc.pas"
     2  .lcomm _V0,16
     3          .text
     4          .def    _P2.0;  .scl    3;      .type   32;     .endef
     5  _P2.0:
     6          pushl   %ebp
     7          movl    %esp, %ebp
     8          subl    $8, %esp
     9          movl    %ecx, -4(%ebp)
```

```
        movl    -4(%ebp), %ecx
        movl    $20, -8(%ebp)
        movl    $25, -4(%ecx)
        leave
        ret
        .def    _Peer_of_p2.1;  .scl   3;     .type  32;     .endef
_Peer_of_p2.1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    %ecx, -4(%ebp)
        movl    -4(%ebp), %ecx
        call    _P2.0
        leave
        ret
        .def    _P3      .3;    .scl   3;     .type  32;     .endef
_P3.3:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    %ecx, -4(%ebp)
        movl    -4(%ebp), %ecx
        movl    -4(%ecx), %ecx
        call    _P2.0
        leave
        ret
        .def    _Another_peer_of_p2.2; .scl  3;  .type  32;  .endef
_Another_peer_of_p2.2:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    %ecx, -4(%ebp)
        leave
        ret
        .def    __p__M0_S0_P1; .scl  3;  .type  32;  .endef
__p__M0_S0_P1:
        pushl    %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        cmpl    $100, 8(%ebp)
        jle     L6
        movl    %ebp, %ecx
        call    _P2.0
        jmp     L1
L6:
        cmpl    $50, 8(%ebp)
        jle     L8
        movl    %ebp, %ecx
        call    _Peer_of_p2.1
        jmp     L1
L8:
        movl    %ebp, %ecx
        call    _Another_peer_of_p2.2
L1:
        leave
        ret
        .section .rdata,"dr"
LC0:
        .ascii "Give Input :\0"
        .text
.globl __p__M0_main_program
        .def    __p__M0_main_program;  .scl  2;  .type  32;  .endef
__p__M0_main_program:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    $1568, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Init
        movl    $-2147483648, 12(%esp)
        movl    $12, 8(%esp)
        movl    $LC0, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_String
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Flush
        cmpl    $0, __p_InOutRes
        je      L12
        call    __p_CheckInOutRes
L12:
        movl    $12, 4(%esp)
        movl    __p_Input, %eax
        movl    %eax, (%esp)
        call    __p_Read_Init
        movl    __p_Input, %eax
        movl    %eax, (%esp)
        call    __p_Read_Integer
        movl    %eax, -8(%ebp)
        movl    %edx, -4(%ebp)
        cmpl    $-1, -4(%ebp)
        jl      L15
        cmpl    $-1, -4(%ebp)
        jg      L16
        cmpl    $-2147483648, -8(%ebp)
        jb      L15
L16:
        cmpl    $0, -4(%ebp)
        jg      L15
        cmpl    $0, -4(%ebp)
        js      L13
        cmpl    $2147483647, -8(%ebp)
        ja      L15
        jmp     L13
L15:
        call    __p_IORangeCheckError
        movl    -8(%ebp), %eax
        movl    %eax, -12(%ebp)
```

```
         jmp     L14
L13:
         movl    -8(%ebp), %eax
         movl    %eax, -12(%ebp)
L14:
         movl    -12(%ebp), %eax
         movl    %eax, _V0
         movl    __p_Input, %eax
         movl    %eax, (%esp)
         call    __p_Read_Line
         cmpl    $0, __p_InOutRes
         je      L19
         call    __p_CheckInOutRes
L19:
         movl    _V0, %eax
         movl    %eax, (%esp)
         call    __p__M0_S0_P1
         leave
         ret
.lcomm _static_ctor_run_condition_1_2,16
.globl __p__M0_init
         .def  __p__M0_init; .scl 2; .type 32; .endef
__p__M0_init:
         pushl   %ebp
         movl    %esp, %ebp
         subl    $8, %esp
         cmpb    $0, _static_ctor_run_condition_1_2
         jne     L20
         movb    $1, _static_ctor_run_condition_1_2
         call    __p_DoInitProc
L20:
         leave
         ret
         .def  ___main;  .scl  2;  .type  32;  .endef
.globl _main
         .def  _main;  .scl  2;  .type  32;  .endef
_main:
         pushl   %ebp
         movl    %esp, %ebp
         subl    $24, %esp
         andl    $-16, %esp
         movl    $0, %eax
         addl    $15, %eax
         addl    $15, %eax
         shrl    $4, %eax
         sall    $4, %eax
         movl    %eax, -4(%ebp)
         movl    -4(%ebp), %eax
         call    __alloca
         call    ___main
         movl    __p_GPC_RTS_VERSION_20050331, %eax
         movl    $0, 12(%esp)
         movl    16(%ebp), %eax
         movl    %eax, 8(%esp)
         movl    12(%ebp), %eax
         movl    %eax, 4(%esp)
```

```
        movl    8(%ebp), %eax
        movl    %eax, (%esp)
        call    __p_initialize
        call    __p__M0_init
        call    __p__M0_main_program
        call    __p_finalize
        movl    $0, %eax
        leave
        ret
        .def    __p_finalize; .scl 3; .type 32; .endef
        .def    __p__M0_init; .scl 3; .type 32; .endef
        .def    __p_initialize;  .scl  3;  .type  32;  .endef
        .def    __p_DoInitProc;  .scl  3;  .type  32;  .endef
        .def    __p_Read_Line;  .scl 3;  .type  32;  .endef
        .def    __p_IORangeCheckError;  .scl  3;  .type  32;  .endef
        .def    __p_Read_Integer;  .scl  3;  .type  32;  .endef
        .def    __p_Read_Init;  .scl  3;  .type  32;  .endef
        .def    __p_CheckInOutRes;  .scl  3;  .type  32;  .endef
        .def    __p_Write_Flush;  .scl  3;  .type  32;  .endef
        .def    __p_Write_String;  .scl  3;  .type  32;  .endef
        .def    __p_Write_Init;  .scl  3;  .type  32;  .endef
```

Consider the situation in case—a, where the enclosing procedure P1 (at lexical level 1) is calling its child procedure P2 (at lexical level 2) at line number 31 of the Listing 6.10. The defining environment for the callee P2 is P1. In this scenario, we find that the caller and the defining environment are the same and hence the caller's activation record serves as the static link for the callee. The caller passes the current frame pointer (the pointer to current activation record) to the callee for using it as a static link. In the x86 assembly language output generated by gpc compiler, corresponding to the program in Listing 6.10, the lines related to the call to P2 from P1 are 51 and 52. We find that the pointer to the current activation record (ebp) is passed to the child routine P2 by moving it to the register ecx in line 51 of the nested_proc.s file and then making the call to the procedure P2.

```
51   movl   %ebp, %ecx    # Moving the pointer to the current activation record to %ecx
52   call   _P2.0         # Calling P2
```

The callee routine P2 assigns 25 to the variable 'v1' defined at lexical level 1, by using the register ecx and indexing to the location where '*v*1' is bound i.e. –4(%ecx) as seen in lines 9 through 12 of nested_proc.s. The line 9 updates the static link field in the current activation record with the value passed by the enclosing procedure in the form of %ecx.

```
 9  movl  %ecx,–4(%ebp)   # Updating the static link field in the current AR
10  movl  -4(%ebp),%ecx   # Fetching it again

12  movl  $25, –4(%ecx)   # Using the Static Link to access 'v1' by indirect addressing
```

The instance of a procedure calling a peer (case—b) can be found at line 16 in the sample program at Listing 6.10, where the procedure 'Peer_Of_P2' calls 'P2' procedure. The defining environment of P2 and Peer_Of_P2 is P1. In this scenario, we find that the callee and the callee share the same the defining environment and hence the static link of the caller can be used as the static link of the callee. The caller passes its own static link stored in its activation record, to the callee to be used as its static link. This is demonstrated by the line 20 through 22 in the x86 assembly language output (nested_proc.s) of the source program, where the current static link in register %ecx is passed again in register ecx to the P2 routine.

```
20   movl   %ecx, -4(%ebp)     # Updating the static link field in the caller AR
21   movl   -4(%ebp), %ecx     # Passing the caller's static link field to the callee
22   call   _P2.0
```

The callee—P2 uses the static link in order to assigns 25 to the variable '*v*1' defined at lexical level 1, by using the register ecx and indexing to the location where '*v*1' is bound, i.e. –4(%ecx) as seen earlier in lines 9 through 12.

The instance of a procedure calling a peer of its ancestor at a lexical level less than the current level (case—c) can be found at line 23 of Listing 6.10, where the procedure P3 calls P2 routine. The defining environment of the P2 is P1. In order to fetch a pointer to the activation record of P1(lexical level 1) in the caller P3(lexical level 3), the static link chain needs to be traversed by 2 levels, i.e. from P3 to its static link Another_Peer_Of_P2() and from it to P1 as shown in Fig. 6.26.

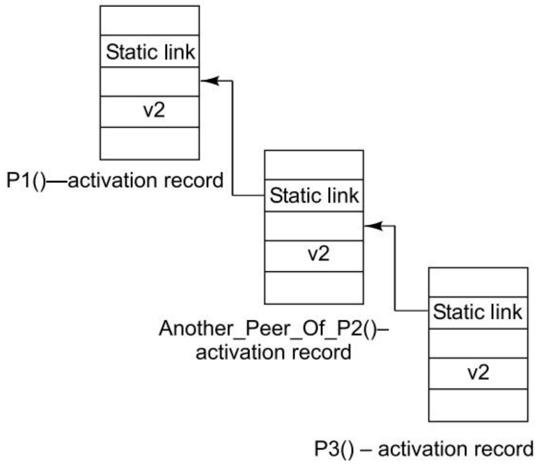


Fig. 6.26 *Traversing the static link chain*

After fetching the pointer to the activation record of P1, it is passed to the callee P2 on the stack similar to the function arguments. This is demonstrated by the lines at 31 through 33.

```
31   movl   -4(%ebp),   %ecx  # Fetching the AR at lexical Level 2 (Another_Peer_Of_P2)
32   movl   -4(%ecx),   %ecx  # Fetching the AR at lexical Level 1 (P1)
33   call   _P2.0
```

The line 31 fetches the static link of the current routine (P3), which would be the pointer to activation record of its enclosing procedure (Another_Peer_Of_P2) at level 2. The line 32 traverses the static link of Another_Peer_Of_P2 to fetch the pointer to activation record of its enclosing procedure at level 1(P1) by doing an indirection. The pointer to activation record of P1 is passed as a static link to the callee routine (P2) in the form of register ecx.

In general, If the current activation is at lexical level 'n' and we are calling a procedure at lexical level 'm' , where m is less than 'n', then we need to traverse (n – m + 1) links to get the static link for passing to the callee procedure. In this example, of procedure P3 (lexical level 3) calling Peer_Of_P2() (lexical level 2) we need to traverse (3 – 2 + 1) = 2 links to get to the activation record of the defining environment. This is passed to the callee to be used as the static link.

The callee—P2 uses the static link in order to assign 25 to the variable '*v*1' defined at lexical level 1, by using the register ecx and indexing to the location where '*v*1' is bound, i.e. –4(%ecx) as seen earlier in lines 9 through 12.

From the discussion, it becomes clear that the calling of an ancestor (case — c) or accessing of a variable defined at a lexical level 'm' from the current level 'n' would require traversing of the access links and will add up to be an overhead in terms of program execution time.

An improved scheme for handling static links defined at various lexical levels is the usage of a data structure called ***display***. A ***display*** is an array of pointers to the activation records. Display[0] contains a pointer to the activation record of the most recent activation of a procedure defined at lexical level 0. Similarly, Display[1] houses a pointer to the activation record of the most recent activation of a procedure defined at lexical level 1, and so on. The number of elements in the display array is given by the maximum level of nesting in the input source program.

In the display scheme of accessing the non-local variables defined in the enclosing procedures, each procedure on activation stores a pointer to its own activation record in the display array at its lexical level. It saves the previous value at that location in the display array and restores it back when the procedure exits. For example, when procedure P2 is entered, it sets the Display[2] to point to its own, i.e. P2 activation record. It saves the previous content of Display[2] and restores it when P2 exits. In this manner, it is ensured that display array always has a pointer to the activation record of the most recent activation of a procedure defined at a particular lexical level. Accessing a variable say 'v_n' defined in the enclosing procedure 'P_n' would involve (a) Fetching the pointer to activation record of P_n by obtaining Display[n] since 'P_n' is at lexical level n and (b) indexing to the memory for the variable 'v_n' within the activation record of 'P_n'.

Consider the implementation of display scheme for the access of non-local variables in the example shown earlier in Listing 6.10. Figure 6.27 shows the display array at the time of activation of P2 when called from P1 represented by the stack layout-1. The display array at the time of activation of P2 when called from Peer_Of_P2 represented by the stack layout-2 is also seen in Fig. 6.27. We can see that in both of these cases the variable '$v1$' can be accessed by (a) fetching the activation record of P1 using display[1] and (b) Indexing into the memory bound to '$v1$' on the stack.

The advantage of the display scheme is that the activation record of any enclosing procedure at lexical level 'n' can be directly fetched using Display [n] as opposed to traversing of the access links in the previous scheme.

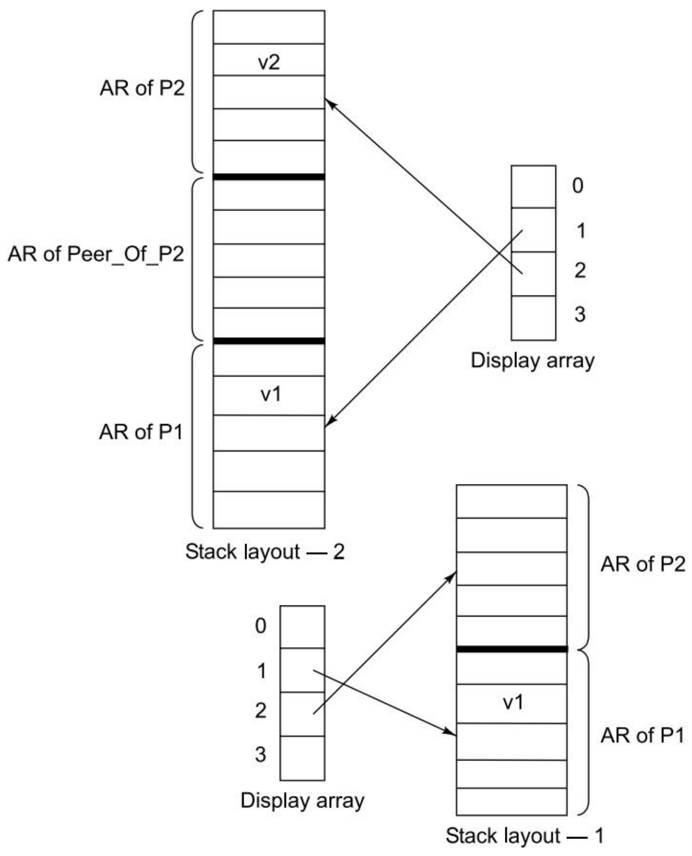


Fig. 6.27 *Using display for non-local access*

6.3.5 Procedure Calling and Return Sequences

Another important aspect in a run-time environment is the calling and return sequences for a procedure execution. In this section, we study about the general steps involved in calling and return sequences. We also examine the division of responsibility between the caller and callee in each of these sequences. In the later part of this section, we also try and understand some of the specifics of calling and return sequences in the C language run-time environment.

Broadly speaking, the generated code for the calling sequence of a function allocates memory for the callee activation record and populates the information into its fields. The return sequence restores the machine state of the caller to the condition it was before the call was made. This enables the caller to continue execution smoothly.

Let's start with trying to understand the calling sequence in a run-time environment. The common operations that need to be performed at the run-time, when a function is called are as follows.

1. The caller evaluates the arguments and stores it in the callee activation record. In the case of run-time environment where the stack is used for housing activation records, this involves pushing the arguments on the stack.
2. The caller stores the return address in the callee activation record and transfers the control to the called function.
3. The callee saves the machine status in the form of contents of the registers so that it can be restored later after its execution.
4. The callee creates the space in its activation record for the local data and initialises it.

The optional control link field in the callee activation field is populated co-operatively by the caller and callee in the case of source languages supporting dynamic scope. The same applies to the optional access link field in cases of source languages supporting nested procedures.

Similar to the calling sequence, when a procedure completes its execution, there are a sequence of actions to be carried out at the run-time as given below.

1. The callee places the return value of the function in its activation record or in some other mutually agreed upon location (e.g. register) from where the caller can pick it up.
2. The callee restores the machine status (in the form of the contents of registers) to the state that was saved in its activation record, during the calling sequence. This enables the caller to resume its execution smoothly. The callee also releases the memory allocated in the activation record for the local variables and compiler-generated temporaries.
3. The caller releases the memory allocated for the parameters in the activation record. With this, the callee activation record is completely released from the memory.
4. The caller copies the return value of the function into its own area.

The target code generator produces code that performs the actions mentioned in the calling and return sequence.

The calling and return sequences that we just studied gives a generic idea of the activities performed. These sequences might vary a little, based on the source language characteristics and the memory organisation at the run-time. The division of responsibility between the caller and callee might also vary a little, depending on the source language characteristics and to a smaller extent on the compiler.

6.3.5.1 Procedure Calling and return Sequences in C Run-time Environment In this section we study some of the specifics of the calling and return sequences in a C run-time environment. The emphasis in this section is on understanding the calling and return sequences in the C runtime environment by studying the x86 assembly code generated for a sample C program. The idea is to have a sample program compiled using the gcc compiler and generate x86 assembly code for the same. We then dissect the generated assembly code to identify the various steps in the calling and return sequences.

The dialog below shows a sample C program being compiled by gcc. We pass the options—save-temps during the compilation for saving the assembly output among others. The assembly output is quoted later for explaining the calling and return sequences.

```
# Program to dislay Calling and return Sequence in C
$ cat -n ex6.c
     1  #include <stdio.h>
     2
     3  void test();
     4  int my_func(int a,int b);
     5
     6  int x,y,z;
     7
     8  int main()
     9  {
    10          test();
    11          return(0);
    12  }
    13
    14  void test()
    15  {
    16          x=5;
    17          y=10;
    18          z=my_func((x+43),(y+20));
    19          x = z + y;
    20          printf("Value of z=%d \n",z);
    21
    22          return;
    23  }
    24
    25  int my_func(int a,int b)
    26  {
    27          int ret;
    28          int c=10,d=20;
    29
    30          c = a *a ;
    31          d = b* b;
    32          ret = c + d + 2 *a *b;
    33
    34          return(ret);
    35  }

# Compiling it for generating the x86 assembly Language code
$ gcc -Wall -c —save-temps ex6.c

# Displaying the x86 assembly file
$ cat -n ex6.s
     1          .file  "ex6.c"
     2          .def   ___main; .scl  2;  .type  32;  .endef
     3          .text
     4  .globl _main
     5          .def  _main;  .scl 2;  .type  32;  .endef
     6  _main:
     7          pushl    %ebp
     8          movl     %esp, %ebp
     9          subl     $8, %esp
    10          andl     $-16, %esp
    11          movl     $0, %eax
    12          addl     $15, %eax
```

```
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        call    _test
        movl    $0, %eax
        leave
        ret
        .section .rdata,”dr”
LC0:
        .ascii “Value of z=%d \12\0”
        .text
.globl _test
        .def    _test;  .scl  2;  .type  32;  .endef
_test:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $5, _x
        movl    $10, _y
        movl    _y, %eax
        addl    $20, %eax
        movl    %eax, 4(%esp)
        movl    _x, %eax
        addl    $43, %eax
        movl    %eax, (%esp)
        call    _my_func
        movl    %eax, _z
        movl    _y, %eax
        addl    _z, %eax
        movl    %eax, _x
        movl    _z, %eax
        movl    %eax, 4(%esp)
        movl    $LC0, (%esp)
        call    _printf
        leave
        ret
.globl _my_func
        .def    _my_func;  .scl  2;  .type  32;  .endef
_my_func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $12, %esp
        movl    $10, -8(%ebp)
        movl    $20, -12(%ebp)
        movl    8(%ebp), %eax
        imull   8(%ebp), %eax
        movl    %eax, -8(%ebp)
        movl    12(%ebp), %eax
        imull   12(%ebp), %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        movl    -8(%ebp), %edx
```

```
69          addl    %eax, %edx
70          movl    8(%ebp), %eax
71          imull   12(%ebp), %eax
72          addl    %eax, %eax
73          leal    (%edx,%eax), %eax
74          movl    %eax, -4(%ebp)
75          movl    -4(%ebp), %eax
76          leave
77          ret
78          .comm   _x, 16   # 4
79          .comm   _y, 16   # 4
80          .comm   _z, 16   # 4
81          .def    _printf;  .scl  3;  .type  32;  .endef
82          .def    _my_func;  .scl  3;  .type  32;  .endef
83          .def    _test;  .scl  3;  .type  32;  .endef
```

We start off by looking at the calling sequence for the function ‘my_func’ in the x86 assembly code generated above for the sample program by the gcc compiler. The function my_func is called from the test() function in line 18 of the ex6.c file. Let’s study the calling sequence for my_func() in a step-by-step fashion.

Step 1: The caller evaluates the arguments and pushes on to the run-time stack before making the function call. In the program above, a call is made to the function ‘my_func’ on the line 18 of the source program ex6.c in the form of z=my_func((x+43),(y+20)). As we can see, the arguments to the function are (x+43) and (y+20). They are evaluated in the caller (function test) and pushed on to the run-time stack in the lines 36 through 41 of the assembly listing (ex6.s) as shown below. Observe that the arguments are pushed on to the stack from right to left, with the rightmost argument pushed first, while the leftmost argument is pushed last on it.

```
36   movl    _y, %eax
37   addl    $20, %eax
38   movl    %eax, 4(%esp)     # Pushing the right most argument

39   movl    _x, %eax
40   addl    $43, %eax
41   movl    %eax, (%esp)      # Pushing the left most argument
```

Step 2: The caller executes the ‘call’ assembly instruction. This pushes the return address on the stack and transfers the control to the called function.

```
42       call    _my_func
```

Step 3: The callee (function my_func) saves the registers and gets the register bp to point to the top of the stack at this point. The BP is used as a base pointer for indexing and accessing local variables as seen in the next step. Note that the saving of machine’s status in this example comprises merely saving the old value of ‘bp’. In situations like say a function called within a loop and others where there is quite some amount of computation going on at the caller site, there would more number of registers saved in the form of machine status.

```
55  _my_func:
56          pushl  %ebp
57          movl   %esp,  %ebp
```

Step 4: The callee creates the local space on the stack and initialises the local data. In the case of the function my_func in the sample code, the size of the local stack is 12 bytes. The size was computed by the code generator depending on the total size of the local variables. The space for the local data is created by simply moving the stack pointer by 12 bytes. The local variable ‘c’ is at the offset –8 from BP. It is being initialised to 10 by the assembly instruction at line 59. Similarly, the variable ‘d’ is at –12 from BP. It is being initialised to 20 by the instruction on line 60. Any access to variable ‘c’ in the function body would resolve to offset –8 from BP as –8(%ebp) and variable ‘d’ to offset –12 from BP as –12(%ebp).

```
58     subl    $12, %esp          # Creating space for Local variables
59     movl    $10, -8(%ebp)      # Initializing Local variable ‘c’
60     movl    $20, -12(%ebp)     # Initializing Local variable ‘d’
```

Let’s look at the above steps in the calling sequence from the perspective of the caller and callee responsibility. Figure 6.28 shows the activation record on the stack along with the demarcation of caller and callee responsibility in the C run-time environment that we just discussed. The caller takes the responsibility of pushing the function arguments and the return address on the run-time stack. The callee takes the responsibility for saving the machine status and also for allocating space for the local variables and the temporaries.

We now look at the return sequence for the same function ‘my_func’ in the x86 assembly code generated for the sample C program by the gcc compiler.

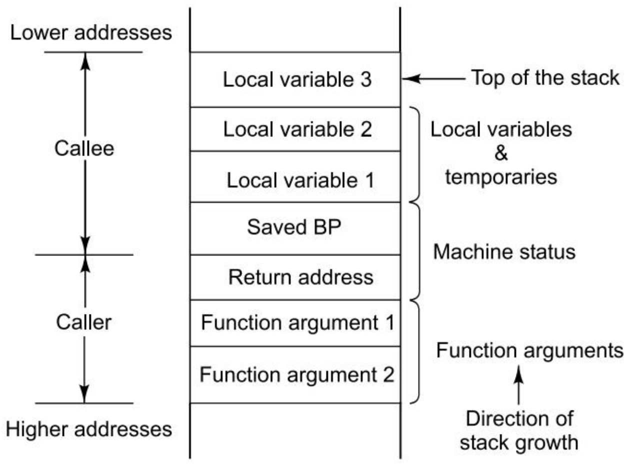


Fig. 6.28 *Stack in C-run-time environment*

Step 1: The callee puts the return value of the function at a location where the caller can pick it up. In the gcc C compiler implementation, the return value of the function is placed in the register eax from where the caller picks it up. The local variable ‘ret’ is situated at offset –4 from BP. The return value is moved from the local variable ‘ret’ to register eax in the line 75 of ex6.s.

```
75        movl        -4(%ebp), %eax
```

Step 2: The callee restores the machine status to the state it was before the function call was made, so that the caller can resume execution. In this case, the value of esp is restored to the value before function started. The value of register ebp is also restored back to the value before the start of the function. This is achieved by the ‘leave’ instruction. By moving back the esp to the value before the function execution, the space allocated for the local variables in step 4 of the calling sequence is also released. The control is transferred to the caller by executing the 'ret' instruction.

```
        leave
        ret
```

Step 3: The caller copies the return value into its own area. The return value, which was stored in register %eax in step 1 of the return sequence, is fetched and copied into the variable 'z'. The assembly instruction corresponding to the copying of the return value into the variable 'z' is shown below.

```
    movl %eax,_z
```

In the cases where the return value of a function is a structure that cannot fit into the register eax, the structure is copied to the caller area by the callee in the gcc C compiler implementation.

The calling and return sequences might vary a little bit with regard to some details in other compiler implementations. The direction in which the stack grows, the locations of heap and stack, and the extent of committed heap and stack are some of the details that might vary depending on the operating system, target architecture and design of the compiler. The main concepts discussed in this section hold good for most of the stack-based C runtime environments.

6.3.6 Parameter-passing Mechanisms

There are several mechanisms by which parameters can be passed to functions, during a function call. The parameter-passing mechanism used in a function call influences the output of the program. The target code generator takes into account the type of the parameter-passing mechanism used in the context and generates code accordingly. An important characteristic of a given runtime environment is the support provided for different parameter-passing mechanisms. In this section we introduce the different ways of passing arguments and the expected program behaviour corresponding to them. We discuss ways of implementing each one of those parameter-passing mechanisms from a code-generation standpoint.

Let's start off by getting the terminology involved in parameter-passing to functions by looking at a sample C program. Consider the small C program shown in Listing 6.11. There is a call made to the function 'max' at line 12. The arguments that are passed to a function at the time of call are called as ***actual parameters***. The variables 'p1' and 'p2' are the actual parameters in the call made to max at line 12. The function definition gives out more details with regard to the number of arguments, type of the arguments, and so on. The function 'max' is defined to take two arguments 'f1' and 'f2', which are integers. The arguments that appear in the function definition are called as ***formal parameters***. The variables 'f1' and 'f2' are the formal parameters for the function 'max'. When formal parameters are defined as a part of a function, they are treated like local variables. For example, in Listing 6.11, the function max is defined from line 16 to 27 with two formal parameters 'f1' and 'f2'. These two are treated like local variables with storage allocated to them in the activation record.

```
1   #include <stdio.h>
2
3   int max(int f1,int f2.);
4   int main()
```

```
{
        int p1,p2;
        int out;

        p1 = 1;
        p2 = 2;

        out = max(p1,p2);
        return(0);
}

int max(int f1,int f2)
{
        int tmp;

        if(f1 > f2){
                tmp=f1;
        }else{
                tmp=f2;
        }

        return(tmp);
}
```

Listing 6.11 *params.c*

In order to understand the different parameter-passing mechanisms one needs to appreciate the difference between a storage location and a value. Consider an assignment statement using an array ‘a’ in C language

```
a[i] = a[j];
```

The expression on the left-hand side, namely, a[i] represents a storage location, while a similar expression on the right-hand side ‘a[j]’ represents a value. An expression used on the left-hand side of the assignment represents a storage location and the one on the right-hand side of the assignment represents a value. We use the term ***l-value***, to refer to the storage location of an expression and ***r-value*** to refer to the value of the expression. The ‘l’ and ‘r’ in the above terms come from the fact that they are on the left or right side of an assignment.

The parameter-passing mechanisms differ on the basis of whether a parameter represents an r-value or an l-value. A consequence of whether the parameter is an l-value or r-value is the net effect of changes made to the formal parameters within the called function. In cases where the parameter represents an l-value, the changes made to the formal parameters in the called function is reflected in the actual arguments at the caller site, i.e. if the called function modifies the formal parameter ‘f1’, that would reflect on the actual parameter ‘p1’. In cases where the parameter represents an r-value, there is no effect of the changes made to the formal parameter ‘f1’ on the actual parameter ‘p1’.

There are 4 different parameter-passing mechanisms that are discussed in this text:

1. Call by value.
2. Call by reference.
3. Call by value-result.
4. Call by name.

Each one of these is discussed in detail in the following sections.

6.3.6.1 Call by Value In the call by value parameter-passing mechanism, the arguments are evaluated at the time of call and they become the values of formal parameters throughout the function. For example, consider the PASCAL program shown in Listing 6.12 in which we use the call by value parameter-passing mechanism for calling the function ‘my_func’ at line number 24. At the time of call, i.e. line 24, the arguments ‘p1’ and ‘p2’ are evaluated, which would yield 4 and 30 in this case. These evaluated values, become the values of the formal arguments ‘f1’ and ‘f2’ during the execution of the function ‘my_func’.

In call by value method, the changes made to the formal parameters are not reflected in the actual arguments at the caller site. In the Listing 6.12, we modify the formal parameters ‘f1’ to 100 and ‘f2’ to 120 at the lines 12 and 13 respectively, but when we print the actual parameters ‘p1’ and ‘p2’ at line 26 after the call to the function ‘my_func’, the modified values are not reflected. The actual arguments ‘p1’ and ‘p2’ continue to have original values, i.e. 4 and 30 even after the call to the function ‘my_func’.

```
 1  PROGRAM sample(input,output);
 2  VAR p1,p2,p3 : integer;
 3
 4          FUNCTION my_func(f1,f2:integer): integer;
 5          BEGIN
 6                if (f1 > f2 )
 7                then
 8                       my_func := f1
 9                else
10                       my_func := f2;
11
12                f1 := 100 ;{ Changing the Value of Formal Parameter }
13                f2 := 120 ;{ Changing the Value of Formal Parameter }
14
15          END;
16
17  BEGIN
18
19          p1 := 4;
20          p2 := 30;
21
22          writeln('Before the function call p1=',p1,' p2=',p2);
23
24          p3 := my_func(p1,p2);
25
26          writeln('After the function call p1=',p1,' p2=',p2);
27
28  END.
```

Listing 6.12 *ex7.pas*

The dialog below shows the compilation and execution of the Pascal program shown in Listing 6.12 that uses the call-by-value mechanism for parameter-passing. The x86 assembly language output for the same program generated by the Pascal compiler—gpc is also seen in the dialog. We will use that to understand the details of implementing the call by value mechanism from a target code generator standpoint. Observing the execution of the program establishes the fact that any changes made to the parameters in a call-by-value method does not have any effect in the actual arguments at the caller site.

```
# Compiling ex7.pas to demonstrate Call-by-Value
$ gpc -Wall --save-temps ex7.pas -o ex7

# x86 assembly language output
$ cat -n ex7.s
     1            .file  "ex7.pas"
     2  .lcomm _P1,16
     3  .lcomm _P2,16
     4  .lcomm _P3,16
     5            .text
     6            .def  __p__M0_S0_My_func;  .scl  3;  .type  32;  .endef
     7  __p__M0_S0_My_func:
     8            pushl    %ebp
     9            movl     %esp, %ebp
    10            subl     $4, %esp
    11            movl     8(%ebp), %eax
    12            cmpl     12(%ebp), %eax
    13            jle      L2
    14            movl     8(%ebp), %eax
    15            movl     %eax, -4(%ebp)
    16            jmp      L3
    17  L2:
    18            movl     12(%ebp), %eax
    19            movl     %eax, -4(%ebp)
    20  L3:
    21            movl     $100, 8(%ebp)
    22            movl     $120, 12(%ebp)
    23            movl     -4(%ebp), %eax
    24            leave
    25            ret
    26            .section .rdata,"dr"
    27  LC0:
    28            .ascii "Before the function call p1=\0"
    29  LC1:
    30            .ascii " p2=\0"
    31  LC2:
    32            .ascii "After the function call p1=\0"
    33            .text
    34  .globl __p__M0_main_program
    35            .def  __p__M0_main_program;  .scl  2;  .type  32;  .endef
    36  __p__M0_main_program:
    37            pushl    %ebp
    38            movl     %esp, %ebp
    39            subl     $24, %esp
    40            movl     $4, _P1
    41            movl     $30, _P2
    42            movl     $1568, 4(%esp)
    43            movl     __p_Output, %eax
    44            movl     %eax, (%esp)
    45            call     __p_Write_Init
    46            movl     $-2147483648, 12(%esp)
    47            movl     $28, 8(%esp)
    48            movl     $LC0, 4(%esp)
    49            movl     __p_Output, %eax
    50            movl     %eax, (%esp)
```

```
        call    __p_Write_String
        movl    $-2147483648, 8(%esp)
        movl    _P1, %eax
        movl    %eax, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Integer
        movl    $-2147483648, 12(%esp)
        movl    $4, 8(%esp)
        movl    $LC1, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_String
        movl    $-2147483648, 8(%esp)
        movl    _P2, %eax
        movl    %eax, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Integer
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Line
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Flush
        cmpl    $0, __p_InOutRes
        je      L6
        call    __p_CheckInOutRes
L6:
        movl    _P2, %eax
        movl    %eax, 4(%esp)
        movl    _P1, %eax
        movl    %eax, (%esp)
        call    __p__M0_S0_My_func
        movl    %eax, _P3
        movl    $1568, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Init
        movl    $-2147483648, 12(%esp)
        movl    $28, 8(%esp)
        movl    $LC2, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_String
        movl    $-2147483648, 8(%esp)
        movl    _P1, %eax
        movl    %eax, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Integer
        movl    $-2147483648, 12(%esp)
        movl    $4, 8(%esp)
        movl    $LC1, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_String
        movl    $-2147483648, 8(%esp)
        movl    _P2, %eax
        movl    %eax, 4(%esp)
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Integer
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Line
        movl    __p_Output, %eax
        movl    %eax, (%esp)
        call    __p_Write_Flush
        cmpl    $0, __p_InOutRes
        je      L4
        call    __p_CheckInOutRes
L4:
        leave
        ret
.lcomm _static_ctor_run_condition_1_2,16
.globl   __p__M0_init
        .def  __p__M0_init;  .scl  2;  .type  32;  .endef
__p__M0_init:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        cmpb    $0, _static_ctor_run_condition_1_2
        jne     L9
        movb    $1, _static_ctor_run_condition_1_2
        call    __p_DoInitProc
L9:
        leave
        ret
        .def    ___main;  .scl  2;  .type  32;  .endef
.globl _main
        .def    _main;  .scl  2;  .type  32;  .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    __p_GPC_RTS_VERSION_20050331, %eax
        movl    $0, 12(%esp)
        movl    16(%ebp), %eax
        movl    %eax, 8(%esp)
```

```
  161          movl     12(%ebp), %eax
  162          movl     %eax, 4(%esp)
  163          movl     8(%ebp), %eax
  164          movl     %eax, (%esp)
  165          call     __p_initialize
  166          call     __p__M0_init
  167          call     __p__M0_main_program
  168          call     __p_finalize
  169          movl     $0, %eax
  170          leave
  171          ret
  172          .def     __p_finalize; .scl   3;     .type   32;     .endef
  173          .def     __p__M0_init;       .scl   3;     .type   32;     .endef
  174          .def     __p_initialize;     .scl   3;     .type   32;     .endef
  175          .def     __p_DoInitProc;     .scl   3;     .type   32;     .endef
  176          .def     __p_CheckInOutRes;  .scl   3;     .type   32;     .endef
  177          .def     __p_Write_Flush;    .scl   3;     .type   32;     .endef
  178          .def     __p_Write_Line;     .scl   3;     .type   32;     .endef
  179          .def     __p_Write_Integer;  .scl   3;     .type   32;     .endef
  180          .def     __p_Write_String;   .scl   3;     .type   32;     .endef
  181          .def     __p_Write_Init;     .scl   3;     .type   32;     .endef

# Executing it
$ ./ex7
Before the function call p1=4 p2=30
After the function call p1=4 p2=30
```

The call-by-value parameter-passing mechanism is implemented by the code generator as follows: (1) The actual arguments are evaluated and their r-values computed at the called site. The r-values of the actual parameters are then 'copied' as the initial values of the formal parameters. (2) The formal parameters in the called function are accessed in a similar manner to the local variables. In the called function, the formal parameters are used for all the computations. Since the called function works on the 'copies' of the actual parameters in the form of formal parameters, any changes done to the formal parameters are not reflected in the actual arguments at the caller site.

The relevant portions of the x86 assembly language code generated by gpc—the Pascal compiler, for Example 7, are shown below. It is annotated to show both of the above aspects in the caller and callee. We have the computing of the r-values of actual arguments and copying them as the initial values of formal parameters in lines 80–84. We see how the arguments are accessed in the called function 'my_func' in the lines 21–22.

```
# Copying the actual parameter 'p2' to the formal parameter 'f2'
80      movl     _P2, %eax
81      movl     %eax, 4(%esp)

# Copying the actual parameter 'p1' to the formal parameter 'f1'
82      movl     _P1, %eax
83      movl     %eax, (%esp)

# Calling the procedure 'my_func'
84      call     __p__M0_S0_My_func

# In the called function
# Accessing the formal parameters similar to the local variables
        movl    $100, 8(%ebp)
        movl    $120, 12(%ebp)
```

The call-by-value method is supported as the default parameter passing mechanism in C and PASCAL language run-time environment.

6.3.6.2 *Call by Reference* In call-by-reference mechanism of parameter passing, the reference (address of the memory location) of the actual parameter is passed to the function instead of the value. In other words, the l-values of the actual parameters are passed to the caller as opposed to the r-values passed in the case of call by value. For example, consider the PASCAL program shown in Listing 6.13 in which we use the call-by-reference parameter-passing mechanism for calling the function 'my_func' at line number 24. This program is identical to the one shown in Listing 6.12 except for the line 4, where we use the keyword VAR at the time of function definition to signify that parameters are passed by reference. At the time of call, i.e. line 22, the address of arguments 'p1' and 'p2' are calculated and passed as arguments. In the called function, 'my_func' any access to the formal parameters 'f1' and 'f2' would be done by using indirect addressing as seen earlier in Section 6.2.3.5.

In call-by-reference method, the changes made to the formal parameters are reflected in the actual parameters at the caller site. This is due to the fact that the addresses of the actual parameters are passed to the caller, and any changes to the formal parameters would be carried out on the same addresses by using the indirect addressing. In Listing 6.13, we modify the formal parameters 'f1' to 100 and 'f2' to 120 at the lines 12 and 13 respectively, when we print the actual parameters 'p1' and 'p2' at line 26 after the call to the function 'my_func', the modified values are reflected. The actual arguments 'p1' and 'p2' have the new values, i.e. 100 and 120 after the call to the function 'my_func' when printed at line 26.

```
 1  PROGRAM sample(input,output);
 2  VAR p1,p2,p3 : integer;
 3
 4     FUNCTION my_func(VAR f1,f2:integer): integer;
 5     BEGIN
 6           if (f1 > f2 )
 7           then
 8                 my_func := f1
 9           else
10                 my_func := f2;
11
12           f1 := 100 ;{ Changing the Value of Formal Parameter }
13           f2 := 120 ;{ Changing the Value of Formal Parameter }
14
15     END;
16
17  BEGIN
18
19     p1 := 4;
20     p2 := 30;
21
22     writeln('Before the function call p1=',p1,' p2=',p2);
23
```

```
    p3 := my_func(p1,p2);

    writeln('After the function call p1=',p1,' p2=',p2);

END.
```

Listing 6.13 *ex8.pas*

The dialog below shows the compilation and execution of the Pascal program shown in Listing 6.13 that uses the call-by-reference mechanism for parameter passing. The x86 assembly language output for the same program generated by the Pascal compiler—gpc is also seen in the dialog. We will use that to understand the details of implementing the call-by-reference mechanism from a target code generator standpoint. Observing the execution of the program establishes the fact that any changes made to the parameters in a call-by-reference method is reflected in the caller.

```
# Compiling ex8.pas to demonstrate Call-by-Reference
$ gpc -Wall —save-temps ex8.pas -o ex8

# x86 assembly language output
$ cat -n ex8.s
     1           .file     "ex8.pas"
     2    .lcomm _P1,16
     3    .lcomm _P2,16
     4    .lcomm _P3,16
     5           .text
     6           .def  __p__M0_S0_My_func;  .scl  3;  .type  32;  .endef
     7    __p__M0_S0_My_func:
     8           pushl  %ebp
     9           movl   %esp, %ebp
    10           subl   $4, %esp
    11           movl   8(%ebp), %eax
    12           movl   12(%ebp), %edx
    13           movl   (%eax), %eax
    14           cmpl   (%edx), %eax
    15           jle    L2
    16           movl   8(%ebp), %eax
    17           movl   (%eax), %eax
    18           movl   %eax, -4(%ebp)
    19           jmp    L3
    20    L2:
    21           movl   12(%ebp), %eax
    22           movl   (%eax), %eax
    23           movl   %eax, -4(%ebp)
    24    L3:
    25           movl   8(%ebp), %eax
    26           movl   $100, (%eax)
    27           movl   12(%ebp), %eax
    28           movl   $120, (%eax)
    29           movl   -4(%ebp), %eax
    30           leave
    31           ret
    32           .section      .rdata,"dr"
    33    LC0:
          .ascii “Before the function call p1=\0”
LC1:
          .ascii “ p2=\0”
LC2:
          .ascii “After the function call p1=\0”
          .text
.globl __p__M0_main_program
          .def  __p__M0_main_program;  .scl  2;  .type  32;  .endef
__p__M0_main_program:
          pushl  %ebp
          movl   %esp, %ebp
          subl   $24, %esp
          movl   $4, _P1
          movl   $30, _P2
          movl   $1568, 4(%esp)
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_Init
          movl   $-2147483648, 12(%esp)
          movl   $28, 8(%esp)
          movl   $LC0, 4(%esp)
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_String
          movl   $-2147483648, 8(%esp)
          movl   _P1, %eax
          movl   %eax, 4(%esp)
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_Integer
          movl   $-2147483648, 12(%esp)
          movl   $4, 8(%esp)
          movl   $LC1, 4(%esp)
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_String
          movl   $-2147483648, 8(%esp)
          movl   _P2, %eax
          movl   %eax, 4(%esp)
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_Integer
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_Line
          movl   __p_Output, %eax
          movl   %eax, (%esp)
          call   __p_Write_Flush
          cmpl   $0, __p_InOutRes
          je     L6
          call   __p_CheckInOutRes
L6:
          movl   $_P2, 4(%esp)
          movl   $_P1, (%esp)
          call   __p__M0_S0_My_func
```

```
         movl   %eax, _P3
         movl   $1568, 4(%esp)
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_Init
         movl   $-2147483648, 12(%esp)
         movl   $28, 8(%esp)
         movl   $LC2, 4(%esp)
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_String
         movl   $-2147483648, 8(%esp)
         movl   _P1, %eax
         movl   %eax, 4(%esp)
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_Integer
         movl   $-2147483648, 12(%esp)
         movl   $4, 8(%esp)
         movl   $LC1, 4(%esp)
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_String
         movl   $-2147483648, 8(%esp)
         movl   _P2, %eax
         movl   %eax, 4(%esp)
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_Integer
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_Line
         movl   __p_Output, %eax
         movl   %eax, (%esp)
         call   __p_Write_Flush
         cmpl   $0, __p_InOutRes
         je     L4
         call   __p_CheckInOutRes
L4:
         leave
         ret
.lcomm _static_ctor_run_condition_1_2,16
.globl __p__M0_init
         .def   __p__M0_init;  .scl  2;  .type  32;  .endef
         __p__M0_init:
         pushl  %ebp
         movl   %esp, %ebp
         subl   $8, %esp
         cmpb   $0, _static_ctor_run_condition_1_2
         jne    L9
         movb  $1, _static_ctor_run_condition_1_2
         call  __p_DoInitProc
L9:
         leave
         ret
```

```
            .def    ___main;  .scl  2;  .type  32;  .endef
    .globl _main
            .def   _main;  .scl  2;  .type  32;  .endef
    _main:
            pushl  %ebp
            movl   %esp, %ebp
            subl   $24, %esp
            andl   $-16, %esp
            movl   $0, %eax
            addl   $15, %eax
            addl   $15, %eax
            shrl   $4, %eax
            sall   $4, %eax
            movl   %eax, -4(%ebp)
            movl   -4(%ebp), %eax
            call   __alloca
            call   ___main
            movl   __p_GPC_RTS_VERSION_20050331, %eax
            movl   $0, 12(%esp)
            movl   16(%ebp), %eax
            movl   %eax, 8(%esp)
            movl   12(%ebp), %eax
            movl   %eax, 4(%esp)
            movl   8(%ebp), %eax
            movl   %eax, (%esp)
            call   __p_initialize
            call   __p__M0_init
            call   __p__M0_main_program
            call   __p_finalize
            movl   $0, %eax
            leave
            ret
            .def   __p_finalize;  .scl  3;  .type  32;  .endef
            .def   __p__M0_init;  .scl  3;  .type  32;  .endef
            .def   __p_initialize;  .scl  3;  .type  32;  .endef
            .def   __p_DoInitProc;  .scl  3;  .type  32;  .endef
            .def   __p_CheckInOutRes;  .scl  3;  .type  32;  .endef
            .def   __p_Write_Flush;  .scl  3;  .type  32;  .endef
            .def   __p_Write_Line;  .scl  3;  .type  32;  .endef
            .def   __p_Write_Integer;  .scl  3;  .type  32;  .endef
            .def   __p_Write_String;  .scl  3;  .type  32;  .endef
            .def   __p_Write_Init;  .scl  3;  .type  32;  .endef

# Executing it
$ ./ex8
Before the function call p1=4 p2=30
After the function call p1=100 p2=120
```

In order to implement the call-by-reference, the code generation has to take care of two aspects: (1) The address of the actual arguments needs to be computed at the called site and passed as the arguments. (2) The references to the formal parameters in the called function should be done using indirect addressing.

The relevant portions of x86 assembly language code generated by gpc—The Pascal compiler, for Example 8, are shown below. It is annotated to show both of the aspects mentioned above. We have the computing of the address of actual arguments and passing it as arguments to the function ‘my_func’, by pushing on to the stack in lines 86–88. We see how the arguments are accessed in the called function 'my_func' using the indirect addressing in the lines 25–28.

```
# Copying the ADDRESSES of the actual parameters on to the stack before making the call
86    movl    $_P2, 4(%esp)
87    movl    $_P1, (%esp)
88    call    __p__M0_S0_My_func

# In the called function
# Using the indirect addressing on the passed references to
# affect the change in the actual parameters itself
25    movl    8(%ebp),  %eax
26    movl    $100, (%eax)          # Assigning 100 using INDIRECT Addressing
27    movl    12(%ebp), %eax
28    movl    $120, (%eax)          # Assigning 120 using INDIRECT Addressing
```

In call by reference, if the function call uses an expression like say my_func((p1+5),p2) or simply my_func(4,5) in the above example, then the compiler needs to handle it specially, since there are no addresses associated with these actual arguments. In such cases, the gpc compiler gives out an error indicating incompatibility between the expected argument and the used ones. This forces the user to correct it and call the function with a proper address location. The FORTRAN77 compiler handles these situations differently. It creates a temporary location where the expression is stored and passes that address as the reference.

An advantage in the call-by-reference is that the values that are passed are not 'copied' as in pass-by-value. This helps in improving performance, especially when large structures are passed to a function.

In Pascal programs, the keyword 'var' in the function definition is used to signify that the parameters are passed by reference. In FORTRAN77, call by reference is the only parameter-passing mechanism. In C language, call by reference is achieved by explicitly defining a function take pointers as parameters and at the time of calling, the addresses are passed as parameters using the address of operator—'&'.

6.3.6.3 Call-by-Value-Result In the call-by-value-result, the values of actual parameters are copied to the formal parameters and used in the called procedure. This is similar to the call-by-value parameter-passing mechanism. However, at the time of completion of the called function, the final value of the formal parameter is copied back into the location of the actual argument. This allows the changes made to the formal parameters within the called function be reflected on the actual parameters at the caller site.

ADA programming language uses this as one of the parameter-passing mechanisms, when the parameter is specified using the 'in out' keyword at the procedure definition. Consider an ADA program mytst.adb shown in Listing 6.14 having a procedure 'myproc' using the 'in out' parameter 'b'. The procedure 'myproc' increments the value of 'a' by 20 and stores it in the 'in out' parameter 'b'. The Listing 6.15 shows the program ex9.adb in which a call is made to 'myproc' with the actual parameters as 'x' and 'y'. The values of 'x' and 'y' at the time of call are 10 and 20 respectively. The two programs — ex9.adb and myproc.adb are compiled and linked together to form an executable—ex9.

```
with Ada.Text_IO; use Ada.Text_IO;
package body mytst is
        procedure myproc(a:in integer; b:in out integer ) is
        begin
              b := a + 20 ;
        end myproc;
end mytst;
```

Listing 6.14 *mytst.adb*

```
with mytst;
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada;
procedure ex9 is
x: integer ;
y: integer ;
begin
     x :=10;
     y :=20;
     Text_IO.Put ("Value of 'y' before the procedure call:");
     Integer_Text_IO.Put (y);
     mytst.myproc(x,y);
     Text_IO.New_Line;
     Text_IO.Put ("Value of 'y' after the procedure call:");
     Integer_Text_IO.Put (y);
end ex9;
```

Listing 6.15 *ex9.adb*

The following dialog shows the compilation and linking of the 'mytst.adb' with the file 'ex9.adb'. We can see from the output of the program that the value of 'y' changes after the execution of the procedure 'myproc'. The x86 assembly listing of the code generated for 'mytst.adb' and 'ex9.adb' by the ADA compiler is also shown in the dialog.

```
# Compiling ADA files to demonstrate Call-by-Value-Result
$ gcc -c --save-temps ex9.adb

$ gcc -c --save-temps mytst.adb

# Binding it
$ gnatbind ex9

# Linking it
$ gnatlink ex9

# Executing it
$ ./ex9
Value of 'y' before the procedure call:       20
Value of 'y' after the procedure call:        30

# x86 assembly language output for ex9.adb
$ cat -n ex9.s
   1             .file  "ex9.adb"
   2             .section .rdata,"dr"
   3             .align 4
   4    LC0:
   5             .ascii "Value of 'y' before the procedure call:"
   6             .align 4
```

```
     7  LC1:
     8          .long  1
     9          .long  39
    10          .align 4
    11  LC2:
    12          .ascii “Value of ‘y’ after the procedure call:”
    13          .text
    14  .globl __ada_ex9
    15          .def  __ada_ex9;  .scl  2;  .type  32;  .endef
    16  __ada_ex9:
    17          pushl  %ebp
    18          movl   %esp, %ebp
    19          subl   $24, %esp
    20          movl   $10, -4(%ebp)
    21          movl   $20, -8(%ebp)
    22          movl   $LC0, %eax
    23          movl   $LC1, %edx
    24          movl   %eax, (%esp)
    25          movl   %edx, 4(%esp)
    26          call   _ada__text_io__put__4
    27          movl   _ada__integer_text_io__default_base, %eax
    28          movl   %eax, 8(%esp)
    29          movl   _ada__integer_text_io__default_width, %eax
    30          movl   %eax, 4(%esp)
    31          movl   -8(%ebp), %eax
    32          movl   %eax, (%esp)
    33          call   _ada__integer_text_io__put__2
    34          movl   -8(%ebp), %eax
    35          movl   %eax, 4(%esp)
    36          movl   -4(%ebp), %eax
    37          movl   %eax, (%esp)
    38          call   _mytst__myproc
    39          movl   %eax, -8(%ebp)
    40          movl   $1, (%esp)
    41          call   _ada__text_io__new_line__2
    42          movl   $LC2, %eax
    43          movl   $LC1, %edx
    44          movl   %eax, (%esp)
    45          movl   %edx, 4(%esp)
    46          call   _ada__text_io__put__4
    47          movl   _ada__integer_text_io__default_base, %eax
    48          movl   %eax, 8(%esp)
    49          movl   _ada__integer_text_io__default_width, %eax
    50          movl   %eax, 4(%esp)
    51          movl   -8(%ebp), %eax
    52          movl   %eax, (%esp)
    53          call   _ada__integer_text_io__put__2
    54          leave
    55          ret
    56          .def   _ada__text_io__new_line__2;   .scl   3;   .type   32;   .endef
    57          .def   _mytst__myproc;     .scl    3;    .type   32;   .endef
    58          .def   _ada__integer_text_io__put__2;     .scl    3;    .type  32;  .endef
    59          .def   _ada__text_io__put__4;  .scl  3;  .type   32;   .endef

# x86 assembly language output for myproc.adb
$ cat -n mytst.s
```

```
   1           .file   "mytst.adb"
   2           .comm   _mytst_E, 16   # 1
   3           .text
   4   .globl _mytst__myproc
   5           .def    _mytst__myproc;    .scl  2;    .type  32;    .endef
   6   _mytst__myproc:
   7           pushl   %ebp
   8           movl    %esp, %ebp
   9           movl    8(%ebp), %eax
  10           addl    $20, %eax
  11           movl    %eax, 12(%ebp)
  12           movl    12(%ebp), %eax
  13           popl    %ebp
  14           ret
```

The code generation for passing by value-result involves three steps: (1) The caller copies the actual arguments by value to the stack, similar to the call-by-value parameter-passing mechanism. (2) The callee executes normally and stores the result (i.e. the changed value of the formal parameter) into a common mutually agreed location. (3) The caller picks up the result from the common mutually agreed location to update the actual argument.

Let's dissect the x86 assembly output of the files ex9.adb and mytst.adb to identify and understand the three steps mentioned above. The lines 34 through 37 of the caller routine ex9 in ex9.s (see Fig. 6.29), show the arguments '*y*' and '*x*' being copied on to the stack. This is the illustration of the first step. The second step happens in the called routine 'myproc' at line 12 of mytst.s (Fig. 6.30) where the value of 'in out' variable '*y*' (result) is moved to a register eax, which serves as the common location between the caller and callee. The third step can be found at line 39 of ex9.s (Fig. 6.29), where caller copies the result from the register eax back to the 'in out' actual argument—*y*.

One of the disadvantages of the call-by-value-result is that, there could be an ambiguity in the result, when the same variable is passed as more than one argument, e.g. func(*y*,*y*). In this type of situation, it is unclear as to which value of '*y*' needs to be copied back into the actual argument at the caller site.

```
# Copying actual argument 'y' (in out parameter) on to the stack
  34      movl     -8(%ebp), %eax
  35      movl     %eax, 4(%esp)

# Copying actual argument 'x' on to the stack
  36      movl     -4(%ebp), %eax
  37      movl     %eax, (%esp)

# Making the call to myproc
  38      call     _mytst__myproc

# Copying the result back to 'y' from the register eax
  39      movl     %eax, -8(%ebp)
```

Fig. 6.29 *ex9.s*

```
 6  _mytst__myproc:
 7          pushl   %ebp
 8          movl    %esp, %ebp

 # Using the parameter passed as value ( y = x + 20 )
```

```
        movl    8(%ebp), %eax
        addl    $20, %eax
        movl    %eax, 12(%ebp)

# Copying the 'final' value of 'y' into register eax
        movl    12(%ebp), %eax
        popl    %ebp
        ret
```

Fig. 6.30 *mytst.s*

6.3.6.4 Call by Name

The call-by-name parameter-passing mechanism was first used in Algol60. Some of the modern functional languages like Haskell use this mechanism for parameter passing.

The call-by-name can be characterised as follows:

1. Every call statement is replaced by the body of the called procedure.
2. Each occurrence of a formal parameter in the called method is replaced with the corresponding actual parameter—the actual text of the argument, not its value.
3. The local variables in the procedure are renamed, in case there exists a name clash with the current set of variables.

Let's take an example using C language syntax, to understand call-by-name parameter-passing mechanism. Consider the C program shown in Listing 6.16 containing a 'swap' routine. We try and understand the call-by-name parameter-passing mechanism by pretending to call the 'swap' routine using the call-by-name. The swap routine is called in the main(), as swap(*i*,*x*[*i*]) in line 22;

```
 1  void  swap(a,b)
 2  {
 3          int temp;
 4          temp = a;
 5          a = b;
 6          b = temp ;
 7  }
 8
 9  int x[10];
10  int i;
11
12  int main()
13  {
14
15          i = 1;
16          x[1] = 4;
17          x[2] = 5 ;
18          x[3] = 7 ;
19          x[4] = 9 ;
20
21
22          swap(i,x[i]);
23
24          return(0);
25
26  }
27
```

Listing 6.16 *ex10.c*

When the 'swap' method is called by name, the body of the swap routine would be literally substituted as shown by the shaded portion in Fig. 6.31. We can see in the shaded portion of Fig. 6.31 that the formal argument 'a' is textually substituted with the actual argument '*i*' and formal argument 'b' is textually substituted by the actual argument '*x*[*i*]'. In the call-by-name parameter-passing mechanism, the evaluation of arguments to the function happens at the time it is used as opposed to the time of call for all other parameter-passing mechanisms. For example, if we had called swap(*i*,*x*[*i*]) with say, call-by-value, it would evaluate *i* and *x*[*i*] at the time of call and pass it as arguments within the function. Here in call-by-name, we can see that the *x*[*i*] is getting evaluated in the fourth line of the shaded portion, where it is used. The other argument '*i*' is evaluated in the second line of the shaded portion.

```
void    swap(a, b)
{
        int temp;
        temp = a;
        a = b;
        b = temp ;
}

int x[10];
int i;
int main()
{

        i = 1;
        x[1] = 4;
        x[2] = 5 ;
        x[3] = 7 ;
        x[4] = 9 ;

        {
             int temp;
             temp = i;
             i = x[i];
             x[i] = temp ;
        }

        return(0);

}
```

Fig. 6.31 *The expansion of swap*

Let's analyse the shaded portion of Fig. 6.31 to understand the consequences of calling the swap routine using call-by-name parameter-passing mechanism. Figure 6.32 shows in comments what happens as each of the statements in the textually substituted swap routine gets executed. The net result of executing the entire shaded portion of the code is that $i = 4$ and $x[4] = 1$. The expected net result would have been $i = 4$ and $x[1] = 1$, the swap of values for *i* and *x*[*i*] as one would expect of the swap routine. This kind of unexpected, counterintuitive results made the call-by-name unpopular. In fact, it has been proved that it is impossible to have a correctly working 'swap' routine using call by name.

```
void    swap(a,b)
{
        int temp;
        temp = a;
        a = b;
        b = temp ;
}

int x[10];
int i;

int main()
{

        i = 1;
        x[1] = 4;
        x[2] = 5 ;
        x[3] = 7 ;

        x[4] = 9 ;


        {
                int temp;
                temp = i;       /* temp = 1 */
                i = x[i];               /* i = x[1] which is 4, so i = 4 */
                x[i] = temp ;   /* x[4] = 1 */
        }


        return(0);

}
```

Fig. 6.32 *Annotation for the macro-expansion of swap function*

Even though the call-by-name is best understood by textually substituting the called routine and replacing the formal parameters in the called method with the corresponding actual parameter. The compilers implement the call-by-name mechanism internally by a different means but achieving the same result as that of textual substitution as discussed above. The reason why textual substitution was not used as implementation mechanism is that, when a function changes, all the functions that are calling it by name have to be recompiled so that the textual substitution happens again. The call-by-name is implemented by using parameter less subroutines called 'thunks' that can evaluate l-value or r-value of the actual parameters. For example, the swap routine used in previous discussion would be implemented using two thunks, where thunk1 evaluates '*i*' and thunk2 evaluating *x*[*i*]. These thunks would help evaluate the parameter at the time of access and implement call-by-name.

Even though the call-by-name has not been popular after Algol60, the idea of macro-expansion has been used later in many programming languages like C. The macro-expansion is an ideal solution to situations where the overhead of setting-up procedure activation is more than the execution of actual body of the procedure.

6.3.7 Comparison of Run-time Environments

In this section, we compare the features of run-time environments of some of the popular programming languages like C, FORTRAN77 and Pascal. The idea is to bring out some of the differences in terms of different ways each of these run-time environments function. The run-time environment of a source language is usually designed to suit its features. Let's take, for example, the FORTRAN77 run-time environment. The FORTRAN77 language specifications supported parameter passing by reference, and did not provide for advanced features like recursion, and dynamic memory allocation. Given these features, a completely static run-time environment with no heap and stack was a good fit. In the case of C language, features like dynamic memory, recursion and others, motivated a stack-based run-time environment, which also had heap memory.

Item for Comparison	C	FORTRAN77	Pascal
Memory organisation	Heap, stack, and static memory. There is support for dynamic memory.	Only static memory. There is no dynamic memory in the form of either stack or heap.	Heap, stack and static memory. There is support for dynamic memory.
Activation records	Stored in the stack area. supports recursion.	Stored in the static area. Cannot support recursion.	Stored in the stack area. Supports recursion.
Procedure calling and return sequences	The calling and return sequences for C deal with allocating, populating and release of the activation record.	The calling and return sequence in a static run-time environment do not allocate and free activation record. The creation of activation record is skipped because it is created statically at the compile time itself and stored in the static region.	The calling and return sequences for PASCAL deal with allocating, filling in all the fields and release of the activation record. The calling sequence in the PASCAL environment also deals with populating the access link of the activation record.
Parameter passing	This supports call by value. However, one can use pass a pointer using call-by-value and simulate a call-by-reference.	This supports only call-by-reference.	This run-time environment supports both call-by-value, and call-by-reference. The default is call-by-value. The keyword 'var' is used to indicate to that the parameter is passed using reference.

In general, some of factors that are considered before conceiving a run-time environment for a new language compiler are as follows: (a) Does the language support recursion? (b) Are local variables supported? Are they required to be visible after the procedure is complete? (c) Does the source language support features like pointers that require dynamic memory? (d) What types of parameter-passing mechanisms are required to be supported?

6.4 CODE GENERATION FOR x86

In this section, we look at implementing a code generator that translates the intermediate code (three-address-code format) into target program (x86 assembly program). There are several ways a code generator can be implemented to translate intermediate code into target assembly program. The method chosen here is a straightforward scheme for generation of assembly code over more sophisticated and better counterparts in view of retaining the simplicity and ease of understanding.

The code generator that we examine in this section associates a sequence of one or more x86 assembly instructions to be generated for each type of three-address-code operator. These generated x86 assembly instructions together achieve the intended functionality for the TAC Operator. This kind of code generation, where each of the TAC operators is associated with a pattern of assembly instructions is sometimes referred to as ***template-based code generation***. Let's take an example of a three-address-code statement using the ADD TAC operator and the associated x86 assembly code as shown in Table 6.6 for understanding the template-based code generation.

Table 6.6 *x86 code generation for ADD TAC statement*

TAC operator	**TAC statement**	**x86 Assembly code to be generated by the code generator**	**Comments**

ADD	`x := y + z`	`movl _y, %eax`
		`addl _z,%eax`
		`movl %eax, _x`
			`Assuming that x, y and z are global variables, each 4 bytes wide`

The generated x86 assembly code shown in Table 6.6 for the ADD TAC statement has three x86 assembly language statements doing the following.

- Moving the operand *y* into the register eax. This prepares the operand for performing the add operation.
- Performing the 'add' operation using *z* and an implicit operand from register eax with the result being stored in the register eax.
- Moving the result of the operation (which is in register eax) into the variable *x*.

These three x86 assembly language statements together achieve the functionality of the ADD TAC statement.

In a similar manner, a sequence of x86 assembly instructions is associated with each of the TAC operators that have been defined in the Intermediate language. The generated x86 assembly instructions have to work with the limitation that both the operands cannot be in memory at the same time. They have to intelligently use the registers as a temporary storage points and perform the required operations.

Before we get into the details of translating other TAC statements into x86 assembly, it would be necessary to understand how the global variables like *x*, *y*, and *z* in Table 6.6 are visible in the context of a generated assembly code.

6.4.1 Global Variable Declarations

In the earlier chapter we studied that the global symbol table contains the details of all the global variables that have been declared in the input source. In order to translate the global declarations in the input source into x86 assembly statements, the code generator goes through the symbol table entry by entry and generates assembly statements. A '.comm' directive in the BSS section is generated for a symbol table entry corresponding to un-initialised global variable. A global symbol label in the data segment together with its initial value is generated for a symbol table entry corresponding to initialised global variable. The code generator generates global symbol label with initialising statements for each element in the case of symbol table entry corresponding to an initialised global array declaration. Table 6.7 shows the transformation of initialised, un-initialised global variables and arrays in the input C source into symbol table and consequently into x86 assembly language code.

Table 6.7 *Translation of global variables into x86 assembly code*

Input C code

```
/* Global Vars Begin */

int x;
int y[20]
char a;
char b[40];

int x1=45;
int y1[3]={10,20,30};
char a1 = 45;
char b1[3]={40,50,60};

/* Global Vars end */

...
....
```

Intermediate representation

Name	Size	Offset
x	4	0
y	80	4
a	1	84
b	40	85
x1	4	125
y1	12	129
a1	1	141
b1	3	142
Global Symbol Table		

x86 Assembly code to be generated by the code generator

```
. section bss # Un-Initialized globals

.comm _x,4 # Size of 4 Bytes

.comm _y,80 # Size of 20x4 = 80 Bytes

.comm _a,1 # Size of 1 Byte

.comm _b,40 # Size of 1x40 = 40 Bytes

. section data
.global_x1
_x1:
      long  45;
.global _y1
_y1:
      long  10;
      long  20;
      long  30;
.global _a1
_a1:
      byte  45;
.global _b1
_b1
      byte  40;
      byte  50;
      byte  60;
```

The translation of global variable declarations into x86 assembly statements as shown in Table 6.7 helps in accessing them later in the execution statements (text section) by using the symbolic name. For example, in the text section following the generated x86 assembly code of Table 6.7, we can have statements like mov %eax,_*x* for moving a value in register eax into the memory associated with variable _x. This facility of being able to access the global variables by name (with an underscore as prefix) is used during the translation of TAC statements with global variables as operands, like the one in Table 6.6.

6.4.2 Statements

As mentioned earlier, the template-based code generator sets out to associate a x86 assembly code sequence to be generated for each TAC operator that has been defined in the intermediate language. Table 6.8 shows the translation into x86 assembly instructions for all the 23 operators that were chosen for intermediate language in Chapter 5. Most of the sample TAC statements, considered for translation in Table 6.8 use *x*, *y*, *z* or *i* as operands. For simplicity, we assume that *x*, *y*, *z* and *i* are 4 byte integer-type global variables declared in the input source translated into x86 assembly declarations as discussed in Section 6.4.1. By virtue of that translation, we can use _*x*, _*y*, _*z* and _*i* to reference the memory associated with each of them in the generated x86 assembly code. In the next section (Section 6.4.3), we see how the code generation can be adapted in case the operands are local variables.

Table 6.8 *Translation from TAC to x86 assembly instructions*

#	TAC Operator	Sample TAC	Translated x86 Assembly Code

1	`ASSIGN`	`x := y`	`movl _y,%eax`
			`movl %eax, _x`

2	`ADD`	`x := y + z`	`movl _y, %eax`
			`addl_ z,%eax`
			`movl %eax,_x`

3	`MUL`	`x = y * z`	`movl _y,%eax`
			`imull _z`
			`movl %eax, _x`

4	`DIV`	`x := y / z`	`movl _y,%eax`
			`cltd`
			`idivl _z`
			`movl %eax, _x`

5	`SUB`	`x := y – z`	`movl _y,%eax`
			`subl _z,%eax`
			`movl %eax,_x`

6	`UMINUS`	`x := – y`	`movl _y,%eax`
			`negl %eax`
			`movl %eax,_x`

7	`L_INDEX_ASSIGN`	`x[i]:= y`	`movl _i,%eax`
			`addl _x,%eax`
			`movl _y,(%eax)`

8	`R_INDEX_ASSIGN`	`y:= x[i]`	`movl _i,%eax`
			`addl _x,%eax`
			`movl (%eax),_y`

9	`ADDR_OF`	`x = &y`	`leal _y, %eax`
			`movl %eax, _x`

10	`LBL`	`lbl my_lbl`	`.align 4`
			`my_lbl :`

11	`GOTO`	`goto my_lbl`	`jmp my_lbl`

12	`LT`	`if x < y goto my_lbl`	`movl _y,%eax`
			`cmp %eax,_x`
			`jl my_lbl`

13	GT	`if x > y goto my_lbl`	`movl _y,%eax`
			`cmp %eax,_x`
			`jg my_lbl`

14	LE	`if x <= y goto my_lbl`	`movl _y,%eax`
			`cmp %eax,_x`
			`jle my_lbl`

15	GE	`if x >= y goto my_lbl`	`movl _y,%eax`
			`cmp %eax,_x`
			`jge my_lbl`

16	EQ	`if x == y goto my_lbl`	`movl _y,%eax`
			`cmp %eax,_x`
			`je my_lbl`

17	NE	`if x != y goto my_lbl`	`movl _y,%eax`
			`cmp %eax,_x`
			`jne my_lbl`

18	PROC_BEGIN	`proc_begin my_func 40`	`.align 4`
			`.globl my_func`
			`my_func :`
			`pushl %ebp`
			`movl %esp,%ebp`
			`subl $40,%esp`

19	PROC_END	`proc_end`	`movl %ebp,%esp`
			`popl %ebp`
			`ret`

20	RETURN	`return x`	`movl _x,%eax`
21	RETRIEVE	`retrieve x`	`movl %eax,_x`
22	PARAM	`param x`	`pushl _x`

23	CALL	`call my_func, 8`	`call my_func`
			`addl $8,%esp`

The simplest translation is for the TAC statements using the ASSIGN TAC operator, where the functionality is carried out by temporarily storing the content of variable *y* in register eax before assigning to *x*. This temporary step is required because x86 assembly language instructions do not allow both the operands to be in memory at the same time.

The translation of TAC statements using binary arithmetic operator OP (ADD/SUB/MUL/DIV) in *x* = *y* OP *z* involves generation of three x86 assembly instructions. (a) Moving *y* into register eax. (b) Performing the computation OP with *z* and register eax as operands, such that the result is stored in register eax. (c) Moving the result, which is in register eax into *x*. The translation for the DIV operator has an additional x86 assembly instruction to clear the direction flag (cltd) before performing the division operation. In the translation of unary negation operator (UMINUS), the 'negl' x86 assembly instruction is used.

The translation of TAC statements using L_INDEX_ASSIGN and R_INDEX_ASSIGN TAC operators involves generation of x86 assembly instructions that use the indirect addressing facility. Recall that *x*[*i*] in three address notation refers to the location, which is '*i*' memory units away from the memory pointed to by *x*. The L_INDEX_ASSIGN TAC instruction translates into 3 assembly instructions. The first two of them help in having the register eax loaded with the correct address of the location where the value needs to be stored. The third instruction uses the indirect addressing to store the value into the memory location. The R_ INDEX_ASSIGN TAC instruction is also translated in the same way except that the indirect addressing is used to fetch the value to stored, while the memory location for storing is accessed using direct addressing. The translation of ADDR TAC operator involves the usage of 'lea' (load effective address) x86 assembly instruction, which fetches the address of a given label.

The translation of TAC statement using the LBL operator involves generation of a label in x86 assembly program. An align directive is also generated to fulfil the condition that x86 architecture requires that the labels in the text segment be aligned at a 4-byte boundary. The translation of the TAC statement using the GOTO operator—'goto my_lbl', involves generation of the jmp assembly instruction with the target as my_lbl.

The TAC statements using the comparison TAC operators like LT,GT, etc. are translated into three x86 assembly instructions. The first two instructions help perform the comparison of the two values. The third assembly instruction generated is the conditional jump that is synonymous to the TAC operator. For example, the TAC operator LT (less than) is translated to 'jl' assembly instruction (jump on less than) with the target label as the operand, while the TAC operator LE (less than or equal to) is translated to 'jle' assembly instruction (jump on less than or equal to).

The translation of T AC statements using function-related TAC operators like PROC_BEGIN, PROC_END, RETURN, etc. are geared towards providing a runtime environment that is similar to the one we studied in Section 6.2.3.6. In the translation for PROC_BEGIN, the code generator generates x86 assembly language directives for being able to view the function globally (.global) and aligning the function entry point at a 4-byte boundary (.align 4). Additionally, a function prolog that saves the BP and moves the value of SP into BP is also generated. This enables the use of BP to access the local variables (negative offsets) and formal parameters (positive offsets). The space for local variables is also created in the PROC_BEGIN translation by shifting the SP (subl instruction) using the cumulative size of local variables. We discuss the local variables and the related stack organisation with more details in Section 6.4.3. In the translation for TAC statements using PROC_END operator, the x86 assembly instructions that constitute the epilog of the function are generated. The epilog of the function involves moving the value of BP into SP and restoring the BP value from what was saved on the stack. The final x86 assembly language statement in the PROC_END operator translation is the 'ret' instruction to transfer the control back to the caller. The translation of the TAC statement using RETURN operator having a value to be returned involves moving the return value into a register eax. We are using the convention that the return value is stored in the register eax. The RETRIEVE TAC instruction uses the same convention and fetches the value of eax register into the variable storing the return value of a function. The translation of TAC statements using the PARAM operator involves generation of a pushl assembly instruction for pushing the parameter on the stack. Observe that the PARAM TAC statements are generated in the reverse order of the arguments passed (last argument first). This ensures that the arguments are pushed in the reverse order before calling the function as expected by the convention studied in Section 6.2.3.6 The translation of the TAC statements using CALL TAC operator involves generation two assembly instructions. The first of them is a 'call' x86 assembly instruction, which transfers the control to the function. The second instruction restores the stack to the original state, the way it was prior to pushing the parameters. The restoring of the stack is performed by incrementing the stack (using addl) with the cumulative size of all the parameters pushed earlier (before the call instruction).

6.4.3 Parameters and Local Identifiers

When the Intermediate code is translated to the x86 assembly language, the accesses to the local variables are transformed to the corresponding locations on the stack. These locations are at known offsets (negative) from the memory pointed to by the base pointer register (ebp). The parameters to the function are pushed on to the stack before a 'call' instruction is issued during the translation of PARAM and CALL TAC statements. The parameters are accessed in the body of the called function by using positive offsets from the memory pointed to by ebp. The arrangement for the local identifiers and the parameters is exactly the same way as we saw in Fig. 6.9. The term 'frame offset' is used to refer to the offset of a local variable or parameter from the memory pointed to by ebp on the stack. It follows that the frame offset would be positive for parameters and negative for local variables.

We have studied earlier that the parameters and the local variables are part of the local symbol table-specific to the function. The 'offset' field in the symbol table entry is used for calculating the frame offset in order to access the location identified for the variable or the parameter on the stack.

Let's take an example to understand how the local variables and parameters are resolved by the code generator during the generation of x86 assembly code. Consider the input source, the corresponding symbol table, intermediate code and the generated x86 assembly code shown in Fig. 6.33. The translated assembly code shows the intermediate code statements (in comments) interspersed along with the x86 assembly statements. This helps in following the translation of each of the TAC statement independently. We can see from the generated x86 assembly code in Fig. 6.33 that the space for local variables is allocated on the stack by line 9 of the x86 assembly code (subl $8,%esp) as a part of translation for PROC_BEGIN. The local variables in the generated x86 assembly code are accessed via indirect addressing using the register ebp as depicted in lines 15,16 and 18 of the generated x86 assembly code. The parameters 'a' and 'b' are accessed in lines 11 and 12 of the generated x86 assembly code.

The frame offsets of both the parameters (*a* and *b*) and local variables (*x* and _*t*0) were derived using a frame offset calculating algorithm shown in Algorithm 6.1.

```
sptr is a pointer to the symbol table entry for the local variable/parameter
marker is the offset of the last of parameters in the symbol table
```

```
calc_frame_offset ( sptr)
{
        if(sptr -> offset > marker ) {/* Local variable */

             frame_offset = marker - sptr->offset
        }else { /* Parameter */

             /* 8 is added to take into account for the saved BP and return address */
             frame_offset = marker – sptr->offset + 8 ;

        }
        return(frame_offset);
}
```

Algorithm 6.1 *Frame offset calculation algorithm*

The frame offset calculation algorithm returns the frame offset given the symbol table pointer for the variable or parameter. It uses the offset of the last of parameters as a marker for distinguishing between a local variable and a parameter. The value of marker is 4 for the example shown in Fig. 6.33. The compiler generated temporary variables are also treated in the same way as local variables. The reader can verify how the frame offsets in lines 11, 12, 15, 16 and 18 were derived using Algorithm 6.1 and the local symbol table in Fig. 6.33.

In the generated x86 assembly code shown in Fig. 6.33, the local variables are accessed in lines 15, 16 and 18. The local storage was created earlier in line 9. Observe that the line 9 has been generated as a part of translation of PROC_BEGIN TAC statement. The translation of PROC_BEGIN also contains the function prolog identical to what we saw in Section 6.2.3.6. Likewise, the translation of PROC_END TAC statement contains the function epilog, which reclaims the local storage area. The translation of PROC_BEGIN and PROC_END TAC statements are crucial to creation and reclaiming of the local storage space.

```
int my_add(int a, int b)
{
    int x;

    x = a + b;
     return(x);
}
```

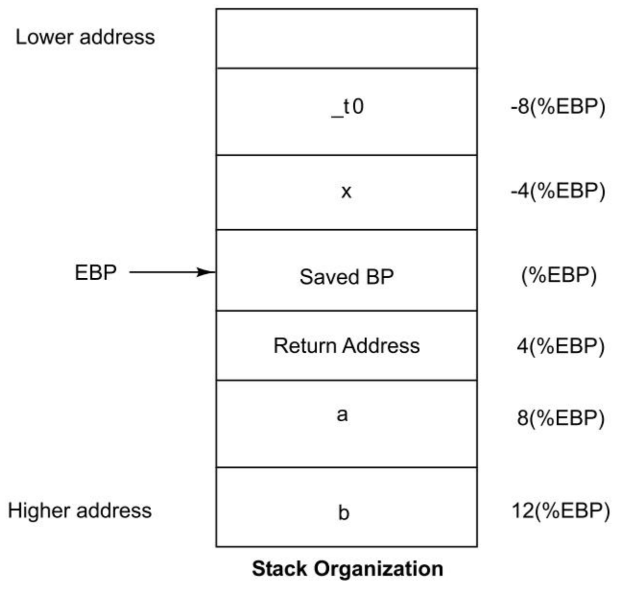
Input source

	Name	Width	Offset
Parameters	a	4	0
	b	4	4
Local variables	x	4	8
	_t0	4	12

Symbol table

```
(0) proc_begin my_add
(1) _t0 := a+b
(2) x:=_t0
(3) return x
(4) goto.L0
(5) label.L0
(6) proc_end my_add
```

Intermediate code



Stack Organization

```
.text

 /* proc_begin my_add */
        .align 4
  .globl _my_add
  _my_add:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
/* _t0 := a + b */
        movl 8(%ebp),%eax
        addl 12(%ebp),%eax
        movl %eax,-8(%ebp)
/* x :=_t0  */
        movl -8(%ebp), %eax
        movl %eax,-4(%ebp)
/* return x */
        movl -4(%ebp),%eax
/* goto.L0  */
        jmp .L0
/* label .L0 */
       .align 4
.L0:
/* proc_end my_add */
        movl %ebp.%esp
        popl %ebp
        ret
```

Generated x86 assembly code

Fig. 6.33 *Local variables and parameters of a function*

6.4.4 Literals

In the course of the input source, we would find statements using numeric literals, e.g. 'x = 30'. There are string literals that are encountered in the input source like say printf("Hello World"). Both the numeric and string literals are stored in a literal table as explained in the earlier chapter.

The code generation for TAC statements that use numeric literals is straightforward. The literal can be embedded as a part of the x86 assembly language statement. Table 6.9 shows the example of a TAC statement using numeric literal and the corresponding generated x86 assembly language code. The numeric literal is accessed using a $ sign as a prefix and it is embedded in the x86 assembly instruction itself.

Table 6.9 *Translation of numeric literals*

TAC operator	**Sample TAC**	**Translated x86 assembly code**
`ASSIGN`	`x := 30`	`movl  $30,_x`

The string literals cannot be embedded in the x86 instruction, they need to be allocated space in the generated x86 assembly program. The code generator goes through all the string literals in the literal table and generates a '.ascii' directive along with the label for each of them to allocate space in the text section itself. The literal is then accessed using the label with a dollar($) prefix in the generated x86 assembly language code. Table 6.10 shows a small input C program, the corresponding TAC statements and the generated x86 assembly code. The lines 2 and 3 in the generated x86 assembly code shows the label and the ascii directive generated for the string literal. The line 13 shows the usage of the string literal with a dollar prefix.

Table 6.10 *x86 code generation for string literals*

Input source	**TAC**	**Translated x86 Assembly Code**

`int printf ()`
 	
`int`
`main ()`
`{`
`    printf ("Hello world\n");`
`}`
	`(0) proc_begin main`
	`(1) param .lc1`
	`(2) call printf 4`
	`(3) retrieve _t0`
	`(4) label .L0`
	`(5) proc_end main`
		`1`
		`2   .text`
		`3   .lc1 :`
		`4   .ascii "Hello world\n\0"`
		`5`
		`6   /* proc_begin main  */`
		`7      .align 4`
		`8   .globl _main`
		`9   _main:`
		`10     pushl %ebp`
		`11     movl %esp,%ebp`
		`12     subl $4,%esp`
		`13  /* param .lc1  */`
		`14     movl $.lc1,%eax`
		`15     pushl %eax`
		`16  /* call printf 4  */`
		`17     call _printf`
		`18     addl $4,%esp`
		`19  /* retrieve _t0  */`
		`20     movl %eax,-4(%ebp)`
		`21  /* label .L0  */`
		`22     .align 4`
		`23  .L0:`
		`24  /* proc_end main  */`
		`25     movl %ebp,%esp`
		`26     popl %ebp`
		`27     ret`

6.5 A TOY C LANGUAGE COMPILER 'MYCC'

This section demonstrates the toy C language compiler (***mycc***), which uses the target code generation concepts described in the previous sections. This compiler includes the lexical analyser, syntax analyser, semantic analyser, and IC generator modules described in the respective chapters. The toy compiler 'mycc' provides support for some of the commonly used features in C. However, it needs improvement to make it into a full-fledged C language compiler offering the complete set of language features.

The toy C compiler—mycc, takes a C source file as an input and generates the corresponding x86 assembly instructions as output. The x86 assembly language instructions are then converted to an executable binary using GNU's assembler and linker. The x86 assembly language output is first converted to an object file (with a .o extension) by using the ***GNU assembler program (as)***. The object file is then converted to an executable (with .exe extension) by using the ***GNU linker program(collect2)*** that comes as a part of gcc.

The dialog below shows 'mycc' taking in different sample C files, and generating the corresponding x86 assembly language instructions. The sample input C files cover some of the important data structures of the C language like arrays, pointer and address operators, structures, and so on. The input c files also include flow of control statements like the if-else, while and switch statements.

The toy C compiler mycc also has an option (*–i*) to generate the intermediate code only without progressing to the code generation stage.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++ -DICGEN -g -Wall -c -o c-small-gram.o c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++ -DICGEN -g -Wall -c -o c-small-lex.o c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$ g++ -DICGEN -g -Wall ic_gen.cc target_code_gen.cc mycc.cc semantic_analysis.cc c-
small-gram.o c-small-lex.o -o mycc.exe

# Input C source - Hello World Program
$ cat -n hello.c
    1
    2  /* Function Prototype */
    3  int printf();
    4
    5  int main()
    6  {
    7        printf("Hello World\n");
    8        return(0); /* to keep the OS Happy */
    9  }

# Compiling it with toy compiler(mycc) to generate x86 assembly language output
$ ./mycc hello.c > hello.s

# Observe the interspersed TAC code (in comments) with the assembly language output
$ cat -n hello.s
    1
```

```
.text
.lc1 :
.ascii “Hello World\n\0”

/* proc_begin main */
      .align 4
.globl _main
_main:
      pushl %ebp
      movl %esp,%ebp
      subl $4,%esp
/* param .lc1 */
      movl $.lc1,%eax
      pushl %eax
/* call printf 4 */
      call _printf
      addl $4,%esp
/* retrieve _t0 */
      movl %eax,-4(%ebp)
/* return 0 */
      movl $0,%eax
/* goto .L0 */
      jmp .L0
/* label .L0 */
      .align 4
.L0:
/* proc_end main */
      movl %ebp,%esp
      popl %ebp
      ret
```

```
# Using gcc in verbose mode to make the executable binary
# Observe the Invoking of the assembler(as) to create an object file
# ..and the linker (collect2) for linking to make the executable
$ gcc -g -v hello.s -o hello.exe
Reading specs from /usr/lib/gcc/i686-pc-cygwin/3.4.4/specs
Configured with: /gcc/gcc-3.4.4/gcc-3.4.4-1/configure --verbose --prefix=/usr --exec-prefix=/usr-
-sysconfdir=/etc --libdir=/usr/lib --libexecdir=/usr/lib --mandir=/usr/share/man --
infodir=/usr/share/info --enable-languages=c,ada,c++,d,f77,java,objc --enable-nls --without-
included-gettext --enable-version-specific-runtime-libs --without-x --enable-libgcj --disable-
java-awt --with-system-zlib --enable-interpreter --disable-libgcj-debug --enable-threads=posix --
enable-java-gc=boehm --disable-win32- registry --enable-sjlj-exceptions --enable-hash-
synchronization --enable-libstdcxx-debug : (reconfigured)
Thread model: posix
gcc version 3.4.4 (cygming special) (gdc 0.12, using dmd 0.125)
  /usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../../i686-pc-cygwin/bin/as.exe --gstabs -o
 /cygdrive/c/WINDOWS/ TEMP/ccS7m1wq.o hello.s
 /usr/lib/gcc/i686-pc-cygwin/3.4.4/collect2.exe -Bdynamic --dll-search-prefix=cyg -o hello.exe
/usr/ lib/gcc/i686-pc-cygwin/3.4.4/../../../crt0.o -L/usr/lib/gcc/i686-pc-cygwin/3.4.4
-L/usr/lib/gcc/i686-pc-cygwin/3.4.4 -L/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../..
/cygdrive/c/WINDOWS/TEMP/ccS7m1wq.o -lgcc -lcygwin -luser32 -lkernel32 -ladvapi32 -lshell32 -lgcc

# Executing the Binary !!
$ ./hello.exe
```

```
Hello World

# mycc has an option -i for merely generating TAC output
$ ./mycc -i hello.c
(0) proc_begin main
(1) param .lc1
(2) call printf 4
(3) retrieve _t0
(4) return 0
(5) goto .L0
(6) label .L0
(7) proc_end main


# Input C file using Local and Global variables
$ cat -n test3.c
    1
    2  /* function prototype */
    3  int printf();
    4
    5  /* Global Variables */
    6  int g_var1,g_var2;
    7
    8  /* Function */
    9  int main()
   10  {
   11        int l_var1,l_var2;
   12
   13        /* Initialization */
   14        g_var1=200;
   15        g_var2=25;
   16
   17        l_var1=g_var1*g_var2;
   18        l_var2 = g_var1/g_var2;
   19
   20        printf("g_var1=%d g_var2=%d l_var1=%d l_var2=%d\n",
   21        g_var1,g_var2,l_var1,l_var2);
   22
   23        return(0);
   24  }

# Compiling it with mycc to generate x86 assembly language output
$ ./mycc test3.c >test3.s

# Using gcc to make the executable binary
$ gcc -g test3.s -o test3.exe

# Executing the Binary
$ ./test3
g_var1=200 g_var2=25 l_var1=5000 l_var2=8

# Input C file having Simple single dimensional array access
$ cat -n test4.c
```

```
     1
     2  /* function prototype */
     3  int printf();
     4
     5  /* Global Variables */
     6  int h;
     7  int a1[20];
     8
     9  /* Function */
    10  int main()
    11  {
    12
    13        /* Initialization */
    14        a1[15]=100;
    15        h=30;
    16
    17        printf("Before : a1[10]=%d h=%d \n",a1[10],h);
    18
    19        /* Array accesses */
    20        a1[10]=h;
    21        h = a1[15];
    22
    23        printf("After : a1[10]=%d h=%d \n",a1[10],h);
    24
    25        return(0);
    26  }

# Compiling it with mycc to generate x86 assembly language output
$ ./mycc test4.c > test4.s

# Using gcc to make the executable binary
$ gcc -g test4.s -o test4.exe

# Executing the Binary
$ ./test4
Before : a1[10]=0 h=30
After : a1[10]=30 h=100

# Input C file using Pointer and Address Operator
$ cat -n test5.c
     1  /* Prototype */
     2  int printf();
     3
     4  int *p;
     5  int x,y;
     6
     7  /* Function */
     8  int main()
     9  {
    10        int tmp;
    11
    12        /* Initialize */
    13        y=10;
    14        x=25;
    15
    16        printf("Before : x=%d y=%d \n",x,y);
    17
    18       /* Move the value of y into x */
    19       p=&x;
    20       tmp = *p;
    21       *p=y;
    22       p = &y;
    23       *p = tmp;
    24
    25       printf("After : x=%d y=%d \n",x,y);
    26
    27       return(0);
    28  }

# Compiling it with mycc to generate x86 assembly language output
$ ./mycc test5.c > test5.s

# Using gcc to make the executable binary
$ gcc -g test5.s -o test5.exe

# Executing the Binary
$ ./test5
Before : x=25 y=10
After : x=10 y=25

# Input C file using Structures
$ cat -n test6.c
     1  /* Prototype */
     2  int printf();
     3
     4  struct my_data
     5  {
     6        int age;
     7        int student_id;
     8  }d1;
     9
    10  int main()
    11  {
    12        /* Initialization */
    13        d1.age=60;
    14        d1.student_id=1234;
    15
    16        printf("Before : age=%d student_id=%d \n",d1.age,d1.student_id);
    17
    18        d1.student_id=4567;
    19        d1.age=20;
    20
    21        printf("After : age=%d student_id=%d \n",d1.age,d1.student_id);
    22
    23        return(0);
    24
    25  }

# Compiling it with mycc to generate x86 assembly language output
$ ./mycc test6.c > test6.s

# Using gcc to make the executable binary
```

```
$ gcc -g test6.s -o test6.exe

# Executing the Binary
$ ./test6
Before : age=60 student_id=1234
After : age=20 student_id=4567

# Input C file using if-else statements
$ cat -n test7.c
     1  /* Prototype */
     2  int printf();
     3
     4  int v1,v2,v3,v4;
     5
     6  int main()
     7  {
     8          v1=10; v2=20; v3=0; v4=0;
     9
    10          printf("Before1 : v1=%d v2=%d v3=%d v4=%d \n",v1,v2,v3,v4);
    11          if(v1 < v2 ){
    12                  v3=10;
    13          }
    14          v4=40;
    15          printf("After1 : v1=%d v2=%d v3=%d v4=%d \n",v1,v2,v3,v4);
    16
    17          v1=20; v2=10; v3=0; v4=0;
    18          printf("Before2 : v1=%d v2=%d v3=%d v4=%d \n",v1,v2,v3,v4);
    19          if(v1 < v2 ){
    20                  v3=10;
    21          }
    22          v4=40;
    23          printf("After2 : v1=%d v2=%d v3=%d v4=%d \n",v1,v2,v3,v4);
    24
    25          return(0);
    26
    27  }

# Compiling it with mycc to generate x86 assembly language output
$ ./mycc test7.c > test7.s

# Using gcc to make the executable binary
$ gcc -g test7.s -o test7.exe

# Executing the Binary
$ ./test7
Before1 : v1=10 v2=20 v3=0 v4=0
After1 : v1=10 v2=20 v3=10 v4=40
Before2 : v1=20 v2=10 v3=0 v4=0
After2 : v1=20 v2=10 v3=0 v4=40

# Input C file using while statement
$ cat -n test8.c
     1  /* Prototype */
     2  int printf();
```

```

int v1,v2;
int v3,v4;

int main()
{

      v1=9; v4=0; v2=1; v3=1;

      while ( v2 <= v1 )
      {
           v3=v3*v2;
           v2 = v2 + 1;
      }
      v4=v3;

      printf(“factorial(%d) = %d \n”,v1,v4);

      return(0);

}

# Compiling it with mycc to generate x86 assembly language output
$ ./mycc test8.c > test8.s

# Using gcc to make the executable binary
$ gcc -g test8.s -o test8.exe

# Executing the Binary
$ ./test8
factorial(9) = 362880
```

SUMMARY

The entity that translates the intermediate code into target program is called as target code generator or simply code generator. The target program can take the form of assembly language instructions, or relocatable machine code or absolute machine code for the processor. The code generator conceives the runtime settings of the program (called as runtime environment) and generates target code accordingly. The important aspects of run-time environment are: (a) memory organisation (b) activation records (c) procedure calling and return sequences (d) parameter passing mechanisms. The code generator discussed here worked on the principle of associating an assembly code skeleton to be generated for each type of TAC operator/statement defined in the intermediate language.

REVIEW QUESTIONS AND EXERCISES

6.1 What does a target code generator do? Explain the various forms of a target program that a target code generator can produce.

6.2 What are the registers available in x86 architecture for a target generator to generate code? Mention the specific uses of each of these registers.

6.3 What is the format of an x86 assembly language program? Describe the different types of statements found in it.

6.4 How are the global variables defined in x86 assembly language? How do you define: (a) A global array of 50 integers of 4 bytes each; (b) a global array of 3 integers of 4 bytes each having initial values of 10, 20 and 30?

6.5 How do you perform arithmetic operations in x86 assembly language? Illustrate with examples.

6.6 How do you (a) fetch the address of a variable; (b) fetch the content of a memory location given its address in x86 assembly language? Can we fetch the content of memory which is at say +20 bytes offset from a given memory address? Illustrate with examples for each.

6.7 How do you define a function in x86 assembly language? How is a stack used for passing the arguments into a function? How do you access the arguments in the body of the function? Illustrate the stack layout during the function body execution showing the function arguments.

6.8 Explain the creation, usage and release of local variables in an x86 assembly function. Illustrate with the stack layout diagram.

6.9 How are the conditional jumps handled in x86 assembly? Illustrate with an example.

6.10 Explain the following terms: (a) activation and lifetime of a procedure (b) Control stack (c) activation tree (d) Binding of a variable to memory.

6.11 What is run-time environment? What are the important elements of runtime environment? How is it controlled in a program that is compiled?

6.12 How is the memory organised in a compiled program? Illustrate the memory organisation in a C run-time environment. Compare it with the memory organisation in FORTRAN77 run-time environment.

6.13 What is an activation record? With the help of a diagram, show the important fields in an activation record.

6.14 Describe the activation record in a C runtime environment? In which section of memory is the activation record located in C runtime environment?

6.15 What is lexical scoping and dynamic scoping in the context of non-local variables? Illustrate with an example.

6.16 How does the C run-time environment handle variables declared in blocks? How does the activation record help in accessing the correct memory for non-local variables?

6.17 How does the activation record help in accessing a non-local variable in a nested procedure of a PASCAL program? Illustrate with an example.

6.18 How are non-local accesses handled in a display scheme? Illustrate with an example.

6.19 What are the procedure calling and returning sequences? Explain the sequence of actions in each of them?

6.20 Illustrate by example, the calling and return sequences in C runtime environment.

6.21 Explain the terms (a) Actual parameters. (b) Formal parameters. Illustrate with an example.

6.22 Explain (a) call-by-reference and (b) call-by-value parameter-passing mechanisms. Illustrate with an example.

6.23 Explain (a) call-by-value-result and (b) call-by-name parameter-passing mechanisms. Illustrate with an example.

6.24 How do the run-time environments of C, PASCAL and FORTRAN77 languages compare against each other?

6.25 Briefly describe the template-based code generation approach. Do you see any performance issues in this approach?

6.26 In semantic analysis, the global variables declared in the input source are converted to symbol table entries. How are the symbol table entries translated into corresponding declarations in x86 assembly language by the target code generator? Illustrate by example for (a) simple variables (b) arrays.

6.27 How do you generate target code for arithmetic TAC operators in a template-based code generation approach? Illustrate with examples for (a) ADD (b) SUB (c) MUL (d) DIV (e) UMINUS.

6.28 How do you generate target code for indexed TAC operators in a template-based code generation approach? Illustrate with examples for (a) L_INDEX_ASSIGN (b) R_INDEX_ASSIGN.

6.29 How do you generate target code for comparison TAC operators in a template-based code generation approach? Illustrate with examples for any three comparison operators.

6.30 How do you generate target code for PROC_BEGIN and PROC_END TAC operators in a template-based code generation approach? Illustrate with examples.

6.31 How are local variables resolved to memory addresses during the translation of intermediate code to target code by the code generator? Draw the stack layout showing the local variables.

6.32 How are numeric and string literals in the intermediate code translated to the target code? Illustrate with an example each.

6.33 State if the following statements are true or false:
- (a) The target code generator takes the intermediate code as the input and generates a target program as output.
- (b) The target program can take one of the three forms (1) assembly language program; (2) relocatable code; (3) absolute code.
- (c) The memory organisation of FORTRAN77 runtime environment does not have a stack or a heap for dynamic memory.
- (d) All the global variables are provided memory in the code section in C run-time environment.

6.34 State if the following statements are true or false:
- (a) The support for nested procedures in PASCAL is accomplished by having a static link in the activation record.
- (b) The term l-value refers to the storage location of an expression and r-value refers to the value of the expression.
- (c) In call-by-reference method, the changes made to the formal parameters are reflected in the actual parameters at the caller site.
- (d) The template-based approach for code generation produces sub-optimal code in terms of performance.

6.35 State if the following statements are true or false:
- (a) In a C runtime environment, the local variables and the function arguments are located on the stack.
- (b) The ‘display’ scheme helps in resolving non-local variable access in the case of nested procedures in PASCAL.
- (c) In ‘call by value-result’ parameter-passing mechanism used in languages like ADA, the final value of the formal parameters is copied back to the actual parameters.
- (d) The literals are accessed using the label with a dollar($) prefix in the generated x86 assembly language code.

CODE OPTIMISATION

7

Introduction

In this chapter, we look at ways of improving the intermediate code and the target code in terms of both speed and the amount of memory required for execution. This process of improving the intermediate and target code is termed as *optimisation*. Section 7.1 demonstrates the fact that there is scope for improving the existing intermediate and target code. Section 7.2 discusses the techniques commonly used to improve the intermediate code. Section 7.3 describes the common methods used in improving the target code generated by the target code generator.

7.1 SCOPE FOR IMPROVEMENT

The correctness of the generated assembly language code is the most critical aspect of a code generator. Also, the efficiency of the generated assembly language code should match closely with the handwritten code, if not be better than it. The code generator that we had discussed in Chapter 6 worked on the principle of statement-by-statement translation of the TAC code into x86 assembly language instruction. This strategy produces correct code, but might not be the most optimal code in terms of efficiency at the run-time.

Consider the sample input source, the corresponding intermediate code and the target code shown in Table 7.1 for understanding the areas of improving the intermediate code and the target code. The intermediate code and the target code have been generated using the toy compiler described in Chapters 5 and 6.

Table 7.1 *Input source, intermediate code and target code*

Input source	Intermediate code	Target code
int a,b;	(0) proc_begin main	1 .comm _a,4
int c,d ;	(1) a := 40	2 .comm _b,4
	(2) b := 20	3 .comm _c,4
int main()	(3) _t0 := a + b	4 .comm _d,4
{	(4) c := _t0	5
a=40;	(5) _t1 := b + c	6 .text
b=20;	(6) d := _t1	7
	(7) label .L0	8 /* proc_begin main */
c = a + b;	(8) proc_end main	9 .align 4
d = b + c;		10 .globl _main
}		11 _main:
		12 pushl %ebp
		13 movl %esp,%ebp
		14 subl $8,%esp
		15 /* a := 40 */
		16 movl $40,_a
		17 /* b := 20 */
		18 movl $20,_b
		19 /* _t0: = a + b */
		20 movl _b,%eax
		21 addl _a,%eax
		22 movl %eax,-4(%ebp)
		23 /* c := _t0 */
		24 movl -4(%ebp),%eax
		25 movl %eax,_c
		26 /* _t1 := b + c */
		27 movl _c,%eax
		28 addl _b,%eax
		29 movl %eax,-8(%ebp)
		30 /* d := _t1 */
		31 movl -8(%ebp),%eax
		32 movl %eax,_d
		33 /* label .L0 */
		34 .align 4
		35 .L0:
		36 /* proc_end main */
		37 movl %ebp,%esp
		38 popl %ebp
		39 ret

Let's analyse the intermediate code to see if any improvements can be made from a run-time efficiency point of view. The quad (3) in the intermediate code uses a temporary *_t0* to store the value of expression '*a* + *b*'. The next quad (quad 4) assigns the temporary *_t0* to the variable '*c*'. The temporary could have been avoided altogether, if the intermediate code is $c := a + b$ instead of quad 3 and 4. In a similar way, the temporary *_t1* could have been avoided by generating $d := b + c$ instead of quads 5 and 6. This proves that there is scope for improvement at the intermediate code level. As the chapter progresses, we can see that there are many more of such opportunities to improve the intermediate code in terms of efficiency. Improving the intermediate code results in reduction of the size of generated target code and overall improvement of run-time efficiency.

Let's analyse the x86 assembly code for improving its runtime efficiency. Line 24 (movl %eax, –4(%ebp)) moves the contents of register eax on to the stack specified by –4(%ebp). The very next x86 assembly statement, line 25 moves the content of stack location –4(%ebp) to register eax (movl –4(%ebp),%eax). Line 24 is redundant, since the register eax already contains the value at –4(%ebp). In a similar fashion, it can be shown that the lines 27 and 31 are also redundant because the destination register already contains the expected value. The generated x86 assembly code could have been better, had the code generation algorithm remembered that the register eax already contained the expected value.

An equivalent handwritten x86 assembly code providing the same functionality is shown in Listing 7.1.

```
.comm   _a,4
.comm   _b,4
.comm   _c,4
.comm   _d,4

# proc_begin main
        .align 4
.globl _main
_main:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        movl $40,_a
        movl $20,_b
        movl _a,%eax
        addl _b,%eax
        movl %eax,_c
        addl _b,%eax
        movl %eax,_d#
label .L0
        .align 4
.L0:
# proc_end main
        movl %ebp,%esp
        popl %ebp
        ret
```

Listing 7.1 *hand_coded_test1.s*

When the output of the code generator is compared with that of the handwritten code, the latter is more compact and efficient (10 assembly instructions) than the former (19 assembly instructions). The code generator should strive to match the efficiency of the handwritten code, if not better than it. In a situation where a piece of code gets executed thousands of times in a loop, a little bit of improvement in the inner loop would make a significant improvement in the overall efficiency of the program.

The intermediate code and the assembly language code generated for the sample input source serves us to understand that there is scope for improvement at both the intermediate code and assembly language levels. The exact nature of improvement in the intermediate code and assembly language outputs varies from input program to program.

In this chapter we discuss the various techniques that can be used for improving intermediate code as well as the assembly language output.

7.2 INTERMEDIATE CODE OPTIMISATION

The intermediate code generated by translation scheme described in Chapter 5, is adequate in terms of correctness with respect to the input program. We saw in the previous section, that there is scope for improving the efficiency of the generated intermediate code in terms of speed of execution and size in memory. In the intermediate code optimisation phase (refer Fig. 1.9), the compiler makes a pass over the generated intermediate code and transforms it into an improved (optimised) form, which is more efficient in terms of speed and size. The transformed intermediate code is then fed to the target code generator for the generation of the target code. In the discussion in Section 7.2.1, we take a look at some of the common transformations made in the intermediate code optimisation phase of the compiler to improve the intermediate code.

7.2.1 Common Sub-expression Elimination

Consider the input source and the corresponding intermediate code in TAC format in Table 7.2. The TAC was generated from the translation scheme explained in Chapter 5. We call the intermediate code shown in Table 7.2 as ***unoptimised intermediate code*** to differentiate it from the version of intermediate code after optimisation using transformations.

Table 7.2 *Input source and the intermediate code*

Input Source	TAC
`int sum_n,sum_n2,sum_n3;`	`(0) proc_begin sum`
`int sum(int n)`	`(1) _t0 := n + 1`
`{`	`(2) _t1 := n * _t0`
`   sum_n = ((n) *(n + 1))/2;`	`(3) _t2 := _t1 / 2`
`   sum_n2=((n)*(n + 1)*(2*n + 1))/6;`	`(4) sum_n := _t2`
`   sum_n3=(((n)*(n + 1))/2)*(((n)*(n + 1))/2);`	`(5) _t3 := n + 1`
`}`	`(6) _t4 := n * _t3`
	`(7) _t5 := 2 * n`
	`(8) _t6 := _t5 + 1`
	`(9) _t7 := _t4 * _t6`
	`(10) _t8 := _t7 / 6`
	`(11) sum_n2 := _t8`
	`(12) _t9 := n + 1`
	`(13) _t10 := n * _t9`
	`(14) _t11 := _t10 / 2`
	`(15) _t12 := n + 1`
	`(16) _t13 := n * _t12`
	`(17) _t14 := _t13 / 2`
	`(18) _t15 := _t11 * _t14`
	`(19) sum_n3 := _t15`
	`(20) label .L0`
	`(21) proc_end sum`

A detailed look at the intermediate code generated in Table 7.2 indicates that the computations made in quads (1) through (3), (12) through (14) and (15) through (17) are essentially the same. These chunks of intermediate code compute the value of the common sub-expression $((n) *(n + 1))/2$, which is used in all the three summations. If we look further, the common sub-expression $((n) *(n + 1))$ is computed 4 times in the statements {1,2 }, {5,6 }, {12,13}, {15,16}. It is possible to optimise the intermediate code to have common sub-expressions computed only once in the function and then re-use the computed values at the second instance.

The optimised version of the intermediate code shown in Table 7.3 evaluates common sub-expression ((*n*) *(*n* + 1)) only once and stores it in _t1 as depicted in statement (2). The variable _t1 is used in statements (3), (6). The sum_*n* is itself a common sub-expression and evaluated only once, since sum_n3 can be looked at sum_n*sum_n. The intermediate code shown in Table 7.3 offers improvement in the speed of execution and also reduces the number of the instructions (memory) compared to the unoptimised IC seen in Table 7.3.

Table 7.3 *Optimised intermediate code*

```
(0) proc_begin sum
(1) _t0 := n + 1
(2) _t1 := n * _t0
(3) sum_n := _t1 / 2
(4) _t5 := 2 * n
(5) _t6 := _t5 + 1
(6) _t7 := _t1 * _t6
(7) sum_n2 := _t7 / 6
(8) sum_n3 := sum_n * sum_n
(9) proc_end sum
```

This process of identifying common sub-expressions and eliminating their computation multiple times in the intermediate code is known as ***common sub-expression elimination***.

7.2.2 Constant Folding

Another common optimisation performed on the intermediate code is known as ***constant folding***. In constant folding, the constant expressions in the input source are evaluated and replaced by the equivalent values at the time of compilation. A constant expression is an expression involving only constants like, say, 4*1, 2*0, and so on. Constant folding improves the speed of execution, since the calculations involving constant expressions are performed at compile time, not at run-time.

Let's take the example of input source and the corresponding intermediate code in TAC format shown in Table 7.4 to understand the constant folding transformation.

Table 7.4 *Input source and the intermediate code*

Input source	TAC
`int arr1[20];`	`(0) proc_begin main`
`int main()`	`(1) _t0 := 0 * 4`
`{`	`(2) _t1 := &arr1`
`        arr1[0]=3;`	`(3) _t1[_t0] := 3`
`        arr1[1]=4;`	`(4) _t2 := 1 * 4`
`}`	`(5) _t3 := &arr1`
	`(6) _t3[_t2] := 4`
	`(7) label .L0`
	`(8) proc_end main`

In the quad (1) of the TAC in Table 7.4, the value 0*4 is computed, which is known to be 0 at the compile time itself. Similarly, in statement (4), the value 1*4 is computed, which is known to be 4 at the time of compilation itself.

As an optimising transformation, the constants can be 'folded' and the resultant value computed at the compile time itself in the statements (1) and (4). In statement (1), 0*4 can be folded to 0, and in statement (4), 1*4 can be folded into 4 at the time of compilation itself. The resulting statements from the constant folding are _*t*0=0 for (1) and _*t*2=4 for (4) respectively. In general, the constant operands can be folded in any of the statements containing arithmetic operators like multiplication, addition, subtraction, division, etc. and transformed into an assignment statement. Table 7.5 shows the optimised IC after using constant folding transformation.

Table 7.5 *Optimised intermediate code*

```
(0) proc_begin main
(1) _t0: = 0
(2) _t1: = &arr1
(3) _t1[_t0]:= 3
(4) _t2: = 4
(5) _t3: = &arr1
(6) _t3[_t2]:= 4
(7) label .L0
(8) proc_end main
```

Constant folding moves the computations (like multiplication, division, addition, subtraction, etc.) involving constants from being computed at the run-time to the compile time, thereby improving the efficiency of the program.

7.2.3 Copy Propagation and Dead Store Elimination

Copy propagation is another commonly used transformation in order to improve the intermediate code. In copy propagation, the use of the variable '*y*' instead of '*x*' is propagated in the statements following a copy statement *x*=*y*.

Let's see how copy propagation works by taking the sample intermediate code shown in Table 7.5. There are two assignment statements (also called as copy statements) that are of interest in the intermediate code of Table 7.5 from the copy propagation standpoint. They are:

(a) The assignment statement (1) where the temporary variable _*t*0 is assigned 0.
(b) The assignment statement (4), where the temporary variable _*t*2 is assigned 4.

The use of value 0 can be propagated in the place of _*t*0 in the statements following the assignment at statement (1). In other words, the variable _*t*0 can be replaced with 0 in statement (3). Similarly, the variable _*t*2 can be replaced with the value 4 in statement (6) following the assignment at statement (4). The resultant intermediate code after the copy propagation is shown in Table 7.6. The statement (3) where _*t*0 has been replaced with 0 and statement (6) where _*t*2 has been replaced with 4 have been shaded in Table 7.6.

By itself, copy propagation does not vastly improve the quality of intermediate code. The intermediate code shown in Table 7.6 is not vastly superior to the IC in Table 7.5. However, copy propagation facilitates other optimising transformations to be performed on the resultant intermediate code. We will now see how copy propagation facilitates an optimising transformation called ***dead store elimination*** to be performed on the resultant code.

Table 7.6 *Intermediate code after copy propagation*

```
(0) proc_begin main
(1) _t0 := 0
(2) _t1 := &arr1
(3) _t1[0] := 3
(4) _t2 := 4
(5) _t3 := &arr1
(6) _t3[4] := 4
(7) label .L0
(8) proc_end main
```

In the intermediate code shown in Table 7.6, the assignment statement (1) can be eliminated, because _*t*0 is no longer used in any of the statements following the assignment. Similarly, the assignment statement (4) can also be eliminated, since _*t*2 is no longer used in any of the statements following the assignment. The values of both _*t*0 and _*t*1 have been copy propagated earlier. The statements (1) and (4) are examples of ***dead store***—statements that compute values which are not used in the program. Dead store can be eliminated from the intermediate code, since it has no effect on the result of the program. The resultant intermediate code after the elimination of assignment statements (1) and (4) is shown in Table 7.7.

Table 7.7 *Intermediate code after elimination of dead stores*

```
(0) proc_begin main
(1) _t1: = &arr1
(2) _t1[0]: = 3
(3) _t3: = &arr1
(4) _t3[4]: = 4
(5) label .L0
(6) proc_end main
```

The dead store elimination improves the speed of execution because we have lesser instructions to execute at the run-time. The dead store elimination also reduces the amount of memory required for storing the code, since it eliminates a few instructions.

In the above example, during the copy propagation, the use of constant 0 was propagated in the place of _*t*0 and the constant 4 was propagated in the place of _*t*2. This kind of copy propagation is sometimes referred to as ***constant propagation*** owing to the propagation of the use of a constant instead of a variable. It is fairly easy to imagine that constant propagation can also facilitate constant folding optimisation.

It is also possible to propagate the use of another variable instead of the existing one in copy propagation. This is known as ***variable propagation***. Let's take a sample input source shown in Table 7.8 for the illustration of variable propagation.

Table 7.8 *cp_prop.1.c*

```
int func(int a,int b,int c)
{
   int d,e,f;

   d = a;


   if(a > 10){
      e = d + b;
   }else{
      e = d + c;
   }

   f = d*e ;

   return(f);
}
```

The corresponding intermediate code at various stages, i.e. unoptimised IC, the IC after copy propagation and then the IC after dead store elimination are all shown in Table 7.9.

Table 7.9 *Intermediate code*

`(0) proc begin func`	`(0) proc_begin func`	`(0) proc_begin func`
`(1) d := a`	`(1) d: = a`	`(1) if a > 10 goto .L0`
`(2) if a > 10 goto .L0`	`(2) if a > 10 goto .L0`	`(2) goto .L1`
`(3) goto .L1`	`(3) goto .L1`	`(3) label .L0`
`(4) label .L0`	`(4) label .L0`	`(4) e: = a + b`
`(5) e := d + b`	`(5) e := a + b`	`(5) goto .L2`
`(6) goto .L2`	`(6) goto .L2`	`(6) label .L1`
`(7) label .L1`	`(7) label .L1`	`(7) e: = a + c`
`(8) e := d + c`	`(8) e := a + c`	`(8) label .L2`
`(9) label .L2`	`(9) label .L2`	`(9) f: = a * e`
`(10) f := d * e`	`(10) f := a * e`	`(10) return f`
`(11) return f`	`(11) return f`	`(11) goto .L3`
`(12) goto .L3`	`(12) goto .L3`	`(12) label .L3`
`(13) label .L3`	`(13) label .L3`	`(13) proc_end func`
`(14) proc_end func`	`(14) proc_end func`	
(A) Unoptimised IC	**(B) IC after copy propagation**	**(C) IC after dead store elimination**

The assignment statement (1) in the unoptimised IC (column A) is of interest from a copy propagation standpoint. The use of variable ‘a’ can be propagated in the place of ‘d’ following the assignment at statement (1). The column (B) in Table 7.9, shows the intermediate code after the variable ‘a’ is used in the place of ‘d’ at statements (5), (8) and (10).

After copy propagation, the assignment statement (1) can be eliminated as a part of dead store elimination, since ‘d’ is no longer used in any of the statements. The intermediate code after the elimination of dead store is shown in column C of Table 7.9.

7.2.4 Dead Code Elimination

In Section 7.2.3, we saw how copy propagation helps eliminate dead stores in the program. In this section, we can see how it helps in eliminating code that is never executed by the program (termed as ***dead code***). The elimination of such dead code reduces the memory required by the program.

Consider the input source shown in Table 7.10 to understand the idea of dead code elimination. In the input source, line 11 makes the check to see if debug is 1 and the line 12 prints the arguments if the check returns true.

Table 7.10 *Dead code in the input source*

```
1   int printf();
2
3   int debug;
4
5   int func(int a,int b,int c)
6   {
7       int v1,v2,v3;
8
9       debug=0;
10
11      if(debug == 1)
12          printf("a=%d b=%d c=%d \n",a,b,c);
13
14      v1=a + b + c;
15
16      return(v1);
17  }
```

Table 7.11 shows the intermediate code resulting from the input source of Table 7.10. In the unoptimised code (column A), there is an opportunity to do copy propagation of the assignment at statement (1). In copy propagation, the use of ‘debug’ is replaced with 0 in the statement (2), where ‘debug’ is used. The copy propagation transform yields the intermediate code shown in the column B of Table 7.11.

In the intermediate code seen at column B, the test in statement (2), i.e. 0 == 1, always returns false. This implies that the control cannot flow to label .L0 from statement (2). There is no other way control can flow to label .L0 either. This makes the statements (4) through (10) in column (B) as dead code. The test 0 == 1 in statement (2) itself becomes redundant, since the result is already known to be false. Hence statement (2) can also be removed as part of dead code elimination. The statement (1) debug := 0, cannot be eliminated in the dead code elimination process, because ‘debug’ is a global variable. The optimised code after elimination of the dead code is shown in column (C) of Table 7.11.

Table 7.11 *The intermediate code*

(A) Unoptimised IC	(B) IC after copy propagation	(C) IC after dead code elimination
(0) proc_begin func	(0) proc_begin func	(0) proc_begin func
(1) debug: = 0	(1) debug: = 0	(1) debug := 0
(2) if debug == 1 goto .L0	(2) if 0 == 1 goto .L0	(2) goto .L1
(3) goto .L1	(3) goto .L1	(3) label .L1
(4) label .L0	(4) label .L0	(4) _t1 := a + b
(5) param c	(5) param c	(5) _t2 := _t1 + c
(6) param b	(6) param b	(6) v1 := _t2
(7) param a	(7) param a	(7) return v1
(8) param .lc1	(8) param .lc1	(8) goto .L2
(9) call printf 16	(9) call printf 16	(9) label .L2
(10) retrieve _t0	(10) retrieve _t0	(10) proc_end func
(11) label .L1	(11) label .L1	
(12) _t1 := a + b	(12) _t1 := a + b	
(13) _t2 := _t1 + c	(13) _t2 := _t1 + c	
(14) v1 := _t2	(14) v1 := _t2	
(15) return v1	(15) return v1	
(16) goto .L2	(16) goto .L2	
(17) label .L2	(17) label .L2	
(18) proc_end func	(18) proc_end func	

Table 7.12 shows another common scenario when dead code exists in the input source during the debugging phase. In this source code, a part of the function has been disabled for debugging reasons. The control returns from the function at the line 9, it cannot reach lines 12,13 or 14 in the input source. The source lines 12,13,14 can be viewed as dead in the input source itself.

We can observe in the unoptimised intermediate code (column B) that the corresponding statements (7) through (12) are dead, since there is no way control can reach them. The dead code is eliminated in the optimised IC shown in column (C) of Table 7.12.

Table 7.12 *Dead code due to debugging code*

(A) Input source	(B) Unoptimised IC	(C) Optimised IC after dead code elimination
1	(0) proc_begin func	(0) proc_begin func
2 int func(int a,int b,int c)	(1) _t0: = a + b	(1) _t0 := a + b
3 {	(2) v1 := _t0	(2) v1 := _t0
4 int v1,v2,v3;	(3) _t1 := v1 / c	(3) _t1 := v1 / c
5	(4) v2: = _t1	(4) v2 := _t1
6 v1 = a + b;	(5) return v2	(5) return v2
7 v2 = v1/c;	(6) goto .L0	(6) goto .L0
8	(7) _t2 := v1 + v2	(7) label .L0
9 return(v2);	(8) v3 := _t2	(8) proc_end func
10	(9) _t3 := v3 + 1	
11 /* Dead Code */	(10) v2 := _t3	
12 v3=v1 + v2;	(11) return v2	
13 v2=v3 + 1;	(12) goto .L0	
14 return(v2);	(13) label .L0	
15 }	(14) proc_end func	

7.2.5 Algebraic Transformations

The quality of the intermediate code can be improved by taking advantage of algebraic identities. An algebraic identity is a relation that holds true for all values of the symbols involved in it. Some of the common algebraic identities that can be used to improve the intermediate code are shown in Table 7.13.

Table 7.13 *Algebraic identities*

Name of the identity	**Example**
Additive Identity	x + 0 = x
Multiplicative Identity	x * 1 = x
Multiplication with 0	x * 0 = 0

The algebraic identity is typically applied on a single intermediate code statement and transformed to a copy statement. Some of the examples are shown in Table 7.14.

Table 7.14 *Algebraic transformations*

IC statement	**Identity applied**	**IC statement after transformation**
y := x + 0	Additive Identity	y := x
y := x * 1	Multiplicative Identity	y := x
y := x * 0	Multiplication with 0	y := 0

The amount of computation is reduced when an algebraic transformation is applied. For example, when an ADD IC statement say *y* := *x* + 0 is replaced by a transformed ASSIGN IC statement *y* := *x*, there is savings in terms of speed, since no addition is involved.

In algebraic transformations, IC statements with operators like ADD and MUL are transformed into copy statements as illustrated in Table 7.14. The copy statements lend well for copy propagation and subsequent dead store/code elimination transformations, which lead to fewer IC statements. The reduction in the IC statements leads to improvement in speed of execution and lower consumption of memory as well.

Let's look at an example, where the unoptimised IC generated contains statements on which algebraic transformations are applied resulting in improvement in the quality of the intermediate code. Consider the C language input source shown in Table 7.15.

Table 7.15 *Input source*

```
1   struct my_struct
2   {
3       int f1[20];
4       int f2;
5   } xyz;
6
7   int func(int index)
8   {
9       xyz.f1[index]=34;
10  }
```

The unoptimised intermediate code generated for the input source in Table 7.15 is shown in column (A) of Table 7.16.

Table 7.16 *Intermediate code.*

`(0) proc_begin func`
`(1) _t0 := &xyz`
`(2) _t1 := 0`
`(3) _t2 := index * 4`
`(4) _t1 := _t2 + _t1`
`(5) _t0[_t1] := 34`
`(6) label .L0`
`(7) proc_end func`
	`(0) proc_begin func`
	`(1) _t0 := &xyz`
	`(2) _t2 := index * 4`
	`(3) _t1 := _t2 + 0`
	`(4) _t0[_t1] := 34`
	`(5) label .L0`
	`(6) proc_end func`
		`(0) proc_begin func`
		`(1) _t0 := &xyz`
		`(2) _t2 := index * 4`
		`(3) _t1 := _t2`
		`(4) _t0[_t1] := 34`
		`(5) label .L0`
		`(6) proc_end func`
			`(0) proc_begin func`
			`(1) _t0 := &xyz`
			`(2) _t2 := index * 4`
			`(3) _t0[_t2] := 34`
			`(4) label .L0`
			`(5) proc_end func`

(A) Unoptimised IC	**(B) Optimised IC after copy propagation & dead code elimination**	**(C) Optimised IC after applying additive identity**	**(D) Optimised IC after copy propagation & dead store elimination**

In the unoptimised code (column A) of Table 7.16, there is an opportunity to do copy propagation following the assignment at statement (2), i.e. (_t1=0). The use of '_t1' can be replaced with 0 in the statement (4). Notice that _t1 cannot be replaced 0 in the statement (5), because _t1 is re-evaluated in statement (4). After the copy propagation, the copy statement (2) is dead and can be eliminated. The intermediate code after copy propagation and dead code elimination is shown in column (B) of Table 7.16.

In the IC after copy propagation in column B, we can apply the additive identity on the statement 3 (shaded) and transform it to a copy statement _t1 = _t2. The transformed IC after applying the additive identity transformation is shown in column (C) of Table 7.16.

The IC in column (C) offers an opportunity to perform copy propagation following the assignment statement (3). We can observe that it is the same assignment statement that was borne out of transformation using the additive identity. The use of '_*t*1' can be replaced with _*t*2 in the statement (4). Following the copy propagation, the copy statement at (3) becomes dead store and hence can be eliminated. The resultant code after copy propagation followed by dead store elimination is shown in (D).

From the above example, it is clear that the algebraic transformations not only replace expensive operations (like add, mul. etc.) with cheaper ones (assign), but also facilitate other optimisations like copy propagation and subsequent dead store elimination.

7.2.6 Strength Reduction Transformation

On most of the processors, the addition operation takes fewer cycles than the multiplication operation. Similarly, a shift operation takes fewer cycles compared to a multiplication or division operation on most of the processors. Extending the view to the intermediate code level, we can say that the addition operator is less expensive than multiplication operator and shift operator is less expensive than multiplication or division operators.

The idea behind the strength reduction transformations is to identify and replace costly operations by less expensive counterparts to achieve the same effect. For example, in strength of reduction transformation a quad $y := x * 2$ can be replaced by another quad $y := x + x$, which is less expensive but achieves the same effect. Table 7.17 shows some of the common strength reduction transformations.

Table 7.17 *Strength reduction transformations*

Expensive operation	Less expensive equivalant	Comments
`y := x * 2`	`y := x + x`	Addition is less expensive than multiplication
`y := x * 32`	`y := x << 5`	Shift is less expensive than multiply
`y := x / 8`	`y := x >> 3`	Shift is less expensive than divide

The strength reduction transformations provide significant benefits, when applied on quads within a loop, due to the fact that the same instruction is executed multiple number of times. We revisit the strength reduction transformations again during the loop optimisation.

7.2.7 Loop Optimisation

The optimisations in the loop have a good scope for performance improvement, since they get executed over and over many times. In loops, any marginal improvement in performance for a single iteration could turn out to be a big improvement in the overall performance of the program, since a loop can get executed multiple times. A couple of loop related transformations, namely loop invariant code motion transformation and strength reduction on induction variables transformation are explained in this section.

7.2.7.1 *Loop Invariant Code Motion* The statements within a loop that compute values, which do not vary throughout the life of the loop are called ***loop invariant statements***. In loop invariant code motion transformation, the loop invariant statements are identified and moved outside of the loop.

Let's see how loop invariant code motion transformation works, by considering the input source and the corresponding unoptimised intermediate code shown below in Table 7.18.

There are two assignment statements that are of interest in the intermediate code of Table 7.18 from the loop invariant code motion transformation perspective. They are:

(a) The assignment statement (6) where the temporary variable _*t*3 is assigned the value &arr. We know that the value of &arr is a constant throughout the life of the program.

(b) The assignment statement (8), where the temporary variable _*t*5 is assigned the value *n*1**n*2. The values of *n*1 and *n*2 are computed before the loop starts at quads (2) and (3) respectively, they do not change during the loop. The value of *n*1**n*2 computed at quad (8) is a constant throughout the life of the loop.

Table 7.18 *Input source and the intermediate code*

Input Source	TAC
`1   int arr[1000];`	`(0)  proc_begin func`
`2`	`(1)  i := 0`
`3   int func(int a,int b)`	`(2)  n1 := a * b`
`4   {`	`(3)  n2 := a - b`
`5       int i;`	`(4)  label .L0`
`6       int n1,n2;`	`(5)  _t2 := i * 4`
`7`	`(6)  _t3 := &arr`
`8       i=0;`	`(7)  _t4 := _t3[_t2]`
`9`	`(8)  _t5 := n1 * n2`
`10      n1  = a * b ;`	`(9)  if _t4 > _t5 goto .L1`
`11      n2  = a - b ;`	`(10) goto .L2`

12	`(11) label .L1`
13 `while(arr[i] > (n1*n2))`	`(12) i := i + 1`
14 `{`	`(13) goto .L0`
15 `i=i + 1;`	`(14) label .L2`
16 `}`	`(15) return i`
17	`(16) goto .L3`
18	`(17) label .L3`
19 `return(i);`	`(18) proc_end func`
20 `}`	
21	

The loop invariant code motion transformation moves the quads (6) and (8) from within to outside of the loop. The performance of the code improves due to this movement, since the amount of computation within the loop decreases. The intermediate code after the loop invariant code motion transformation is shown in Table 7.19.

Observe that the number of quads to be executed in each of the iteration before the loop invariant code motion transformation was 10 (quad 4 through 13 in Table 7.18). The number of quads to be executed in each of the iteration after the transformation is 8 (quad 6 through 13 in Table 7.19).

The loop Invariant code motion transformation improves the speed of execution because there are lesser instructions to execute in each of the iterations.

Table 7.19 *Intermediate code after loop invariant code motion transformation*

```
(0) proc_begin func
(1) i := 0
(2) n1 := a * b
(3) n2 := a - b
(4) _t3 := &arr
(5) _t5 := n1 * n2
(6) label .L0
(7) _t2 := i * 4
(8) _t4 := _t3[_t2]
(9) if _t4 > _t5 goto .L1
(10) goto .L2
(11) label .L1
(12) i := i + 1
(13) goto .L0
(14) label .L2
(15) return i
(16) goto .L3
(17) label .L3
(18) proc_end func
```

7.2.7.2 ***Strength Reduction on Induction Variables*** The strength reduction transformations can be carried out in any part of the intermediate code. However, the loops offer more returns in terms of performance benefits, on the application of the strength reduction transforms, since the code gets executed multiple times.

An ***induction variable*** is a variable that changes by a fixed quantity on each of the iterations of a loop. Consider the input source and the corresponding unoptimised intermediate code shown in Table 7.20. The variable '*ind*' is a user defined induction variable that increases by 1 on each of the iterations (quad 8). The compiler-generated variable '_*t*0' is another induction variable that increases by 4 on each of the iterations (quad 5). The strength reduction transformations are usually applied on the induction variables in a loop to get substantial performance benefits.

Table 7.20 *Input source and intermediate code*

Input source	TAC

`int ind;`
`int a[20];`
 	
`int func()`
`{`
`    while(ind < 20){`
`        a[ind]=10;`
`        ind=ind + 1;`
`    }`
`}`
	`(0) proc_begin func`
	
	`(1) label .L0`
	`(2) if ind < 20 goto .L1`
	`(3) goto .L2`
	`(4) label .L1`
	`(5) _t0: = ind * 4`
	`(6) _t1: = &a`
	`(7) _t1[_t0]: = 10`
	`(8) ind: = ind + 1`
	`(9) goto .L0`
	
	`(10) label .L2`
	`(11) label .L3`
	`(12) proc_end func`

Table 7.21 shows the intermediate code after the application of reduction of strength on the induction variable _*t*0. The quad assigning the initial value of _*t*0, i.e. _*t*0 := ind *4 is moved out of the loop as the quad (0a). This is used as an initial value, for _*t*0. An additional quad (8a) computing the value of _*t*0 from its previous value i.e. _*t*0: = _*t*0 + 4, is inserted right after the quad 8, which computes the value of main induction variable 'ind'. Observe that the transformed loop in Table 7.21 is functionally equivalent to the intermediate code at Table 7.20.

Table 7.21 *Intermediate code after reduction of strength transformation*

```
(0) proc_begin func
(0a) _t0 := ind * 4
(1) label .L0
(2) if ind < 20 goto .L1
(3) goto .L2
(4) label .L1
(5)
(6) _t1 := &a
(7) _t1[_t0] := 10
(8) ind := ind + 1
(8a) _t0 := _t0 + 4
(9) goto .L0
(10) label .L2
(11) label .L3
(12) proc_end func
```

The strength reduction transformation on induction variables improves the speed of execution because we have less expensive instruction, i.e. addition being substituted for multiplication in each of the iterations.

7.2.8 Introductory Concepts for Implementing Intermediate Code Optimisation

In order to implement the transformations mentioned in the Section 7.2.7, the compiler is required to perform analysis of the intermediate code. Before we get into the details of the analysis performed by the optimiser module, it is essential to familiarise ourselves with some relevant terminology and concepts. This section describes such terminology and concepts.

7.2.8.1 Basic Block The idea of a ***basic block*** is very useful in implementing the optimising transformations on the intermediate code. A basic block is a sequence of intermediate code statements in which the control enters at the beginning and leaves only at the end. Within a basic block control flows sequentially. Branching in flow of control can only happen in the last statement of a basic block.

Consider the input source and its corresponding intermediate code shown in Table 7.22. The input source contains a branching statement (if statement), which is also reflected in the IC. It is possible to break this intermediate code into logical chunks of sequential code called as basic blocks.

Table 7.22 *Input source and the intermediate code*

Input source	Unoptimised TAC
	`(0) proc_begin func`
`int func(int a,int b, int c)`	`(1) _t0 := 2 * a`
`{`	`(2) x := _t0`
`    int x,y,z;`	`(3) _t1 := 2 * a`
	`(4) _t2 := 5 * b`
	`(5) _t3 := _t1 + _t2`
`    x  = 2 * a ;`	`(6) y := _t3`
`    y  = 2 * a + 5 * b ;`	`(7) if a > 1 goto .L0`
	`(8) goto .L1`
`    if(a > 1){`	`(9) label .L0`
`        x  = 2 * a + 3 * b + 20;`	`(10) _t4 := 2 * a`
`        y  = 2 * a + 4 * b + 40 ;`	`(11) _t5 := 3 * b`
`    }`	`(12) _t6 := _t4 + _t5`
	`(13) _t7 := _t6 + 20`
`    z  = x * y ;`	`(14) x := _t7`
	`(15) _t8 := 2 * a`
`    return(z);`	`(16) _t9 := 4 * b`
`}`	`(17) _t10 := _t8 + _t9`
	`(18) _t11 := _t10 + 40`
	`(19) y := _t11`
	`(20) label .L1`
	`(21) _t12 := x * y`
	`(22) z := _t12`
	`(23) return z`
	`(24) goto .L2`
	`(25) label .L2`
	`(26) proc_end func`

Table 7.23 shows intermediate code logically split into basic blocks separated by a dotted line. Each of these basic blocks is given names such as B0, B1, B2, etc. representing basic block 0, basic block 1, and so on.

Table 7.23 *Basic blocks*

#	Intermediate code

B0	`(0) proc_begin func`
	`(1) _t0 := 2 * a`
	`(2) x := _t0`
	`(3) _t1 := 2 * a`
	`(4) _t2 := 5 * b`
	`(5) _t3 := _t1 + _t2`
	`(6) y := _t3`
	`(7) if a > 1 goto .L0`

B1	`(8) goto .L1`

B2	`(9) label .L0`
	`(10) _t4 := 2 * a`
	`(11) _t5 := 3 * b`
	`(12) _t6 := _t4 + _t5`
	`(13) _t7 := _t6 + 20`
	`(14) x := _t7`
	`(15) _t8 := 2 * a`
	`(16) _t9 := 4 * b`
	`(17) _t10 := _t8 + _t9`
	`(18) _t11 := _t10 + 40`
	`(19) y := _t11`

B3	`(20) label .L1`
	`(21) _t12 := x * y`
	`(22) z := _t12`
	`(23) return z`
	`(24) goto .L2`

B4	`(25) label .L2`
	`(26) proc_end func`

We study about the details of how to split a given set of quads into basic blocks, how to optimise within a basic block and other operations on a basic block in the forthcoming sections. At this point, the reader needs to appreciate the idea of a basic block of intermediate code in which control flows sequentially.

7.2.8.2 Directed Acyclic Graph Another concept used extensively in implementing optimising transformations for intermediate code is a directed acyclic graph (DAG). The DAG is a data structure used for implementing optimising transformations on the intermediate code within a basic block. The DAG can be constructed from the three address code statements pertaining to a basic block. A DAG is usually shown in a pictorial fashion. Figure 7.1 shows a basic block of three address code statements and its corresponding DAG.

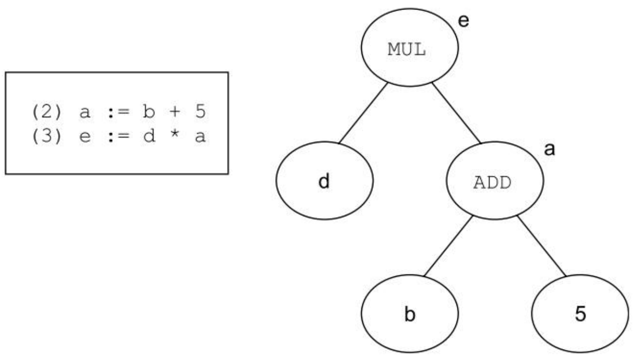
A DAG for a basic block consists of two kinds of nodes, namely,

(1) Leaf nodes

(2) Interior nodes

The leaf nodes are the nodes that do not have children. The leaf nodes are labelled by unique identifiers or constants. Figure 7.1 shows leaf nodes labelled as '*d*', '*b*' and '5'. As a convention in this book, the label for a node is written inside the bubble representing the node.

The interior nodes are the nodes that have children. The children could be either another interior node or a leaf node. The interior nodes are labelled by an operator. Figure 7.1 shows two interior nodes, one of them is labelled with the ADD operator and the other is labelled with a MUL operator. Table 5.2 in Chapter 5 gives a list of all the operators that are considered in the intermediate language that we defined.

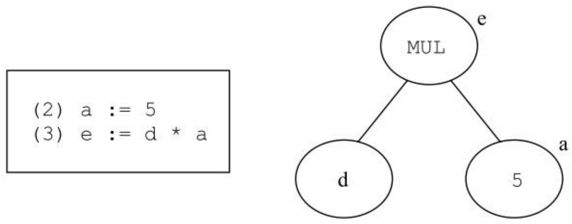


```
(2) a := b + 5
(3) e := d * a
```

Fig. 7.1 *Three address code and its DAG*

The leaf nodes and the interior nodes can have an attached identifier list (shown outside the bubble). The attached identifier list represents the identifiers holding the computed value in the case of interior nodes. For example, in Fig. 7.1, the attached identifier list for the operator node ADD, contains the identifier '*a*'. The identifier '*a*' holds the computed value (*b* + 5). Similarly, the identifier 'e' attached to the node labelled MUL holds the computed value *d* * *a*.

In the case of leaf, the attached identifier holds the value of label. Figure 7.2 shows a DAG having an identifier '*a*' attached to a leaf node labelled 5. The identifier '*a*' holds the value of the label, i.e. 5.



```
(2) a := 5
(3) e := d * a
```

Fig. 7.2 *Leaf node with an attached identifier*

In the later sections, we will study more on the construction of DAG and its usage in optimising IC within a basic block. At this point, it is important to understand that (a) DAG can be constructed from three address code and (b) DAG is used for performing optimising transformations on the intermediate code within a basic block.

7.2.8.3 Local Optimisation and Global Optimisation It is possible to perform optimising transformations like common sub-expression evaluation, copy propagation, etc. that are localised to a basic block. These optimising transformations can be arrived at by analysing the intermediate code of the basic block in isolation. This kind of optimisation in which both the analysis and the transformations are localised to a basic block is known as ***local optimisation***. The transformations in local optimisation are called as ***local transformations***. The name of transformation is usually prefixed with 'local' while referring to the local transformation, e.g. local common sub-expression elimination, local copy propagation, and so on.

Consider the input C language source and the corresponding unoptimised intermediate code shown in Table 7.24 to get an idea of local optimisation.

Table 7.24 *Input C-source and the unoptimised intermediate code*

Input source	**Unoptimised TAC**
1	(0) proc_begin func
2 int func(int a, int b, int c)	(1) _t0 := 2 * a
3 {	(2) x := _t0
4 int x,y,z;	(3) _t1 := 2 * a
5	(4) _t2 := 5 * b
6	(5) _t3 := _t1 + _t2
7 x = 2 * a ;	(6) y := _t3
8 y = 2 * a + 5 * b ;	(7) if a > 1 goto .L0
9	(8) goto .L1
10 if(a > 1){	(9) label .L0
11 x = 2 * a + 3 * b + 20;	(10) _t4 := 2 * a
12 y = 2 * a + 4 * b + 40 ;	(11) _t5 := 3 * b
13 }	(12) _t6 := _t4 + _t5
14	(13) _t7 := _t6 + 20
15 z = x * y ;	(14) x := _t7
16	(15) _t8 := 2 * a
17 return(z);	(16) _t9 := 4 * b
18 }	(17) _t10 := _t8 + _t9
	(18) _t11 := _t10 + 40
	(19) y := _t11
	(20) label .L1
	(21) _t12 := x * y
	(22) z := _t12
	(23) return z
	(24) goto .L2
	(25) label .L2
	(26) proc_end func

Table 7.25 shows the unoptimised TAC broken up into basic blocks. Consider the unoptimised TAC in basic block 0. By analysing the TAC statements of basic block 0, we can conclude that there is a common sub-expression 2**a*, which is computed twice at statement (1) and (3). We can optimise the IC of the block 0 by computing the common sub-expression '2**a*' once at statement (1) and then re-use it. We can eliminate the statement (3) in which we re-compute the common sub-expression 2**a*. Another opportunity for optimisation in the basic block 0 exists at statement (1) and (2), where the unnecessary assignment to the temporary '_*t*0' can be avoided by directly assigning '2**a*' to '*x*'. Column (B) in Table 7.25 shows the resulting optimised code for basic block 0 after applying the optimising transformations mentioned above. Observe that the opportunities for optimisation in basic block 0 were decided by analysing the TAC belonging to the basic block 0 only. This ability to optimise the intermediate code of a basic block by analysing the TAC belonging to its own self is the main characteristic of local optimisation. Table 7.25 shows the locally optimised TAC for all the other basic blocks B1, B2, B3 and B4. The optimised TAC for each of these basic blocks was obtained by analysing the TAC pertaining to that particular basic block only. Thus, in local optimisation, the optimising transformations like common sub-expression elimination, etc. are applied locally to each one of basic blocks independently without taking note of TAC in any other basic block.

Table 7.25 *Local optimisation*

#	(A) Unoptimised code		(B) Locally optimised code	Remarks
B0	(0) proc_begin func (1) _t0 := 2 * a (2) x := _t0 (3) _t1 := 2 * a (4) _t2 := 5 * b (5) _t3 := _t1 + _t2 (6) y := _t3 (7) if a > 1 goto .L0	-->	(0) proc_begin func (1) x := 2 * a (2) _t2 := 5 * b (3) y := x + _t2 (4) if a > 1 goto .L0	In the optimized code for this block (a) The unnecessary assignment to _t0 has been eliminated. (b) The common sub-expression 2*a is computed only once in (1) and re-used (3)
B1	(8) goto .L1	-->	(8) goto .L1	No room for Local optimi-sation in this block
B2	(9) label .L0 (10) _t4 := 2 * a (11) _t5 := 3 * b (12) _t6 := _t4 + _t5 (13) _t7 := _t6 + 20 (14) x := _t7 (15) _t8 := 2 * a (16) _t9 := 4 * b (17) _t10 := _t8 + _t9 (18) _t11 := _t10 + 40 (19) y := _t11	-->	(6) label .L0 (7) _t4 := 2 * a (8) _t5 := 3 * b (9) _t6 := _t4 + _t5 (10) x := _t6 + 20 (11) _t9 := 4 * b (12) _t10 := _t4 + _t9 (13) y := _t10 + 40	In the optimized code for this block (a) The unneccesary assignments to _t7 and _t11 have been eliminated (b) Recomputation of 2*a into _t8 has been avoided, and _t4 is used instead.
B3	(20) label .L1 (21) _t12 := x * y (22) z := _t12 (23) return z (24) goto .L2	-->	(14) label .L1 (15) z := x * y (16) return z (17) goto .L2	In the optimized code for this block (a) The unneccesary assignment to _t12 has been eliminated
B4	(25) label .L2 (26) proc_end func	-->	(18) label .L2 (19) proc_end func	No room for Local Optimization in this block

In contrast to the local optimisation, ***global optimisation*** involves analysis and transformations of the TAC spanning across multiple basic blocks of a procedure. The transformations in global optimisation are called ***global transformations***. The name of transformation is usually prefixed with 'global' while referring to the global transformation, e.g. global common sub-expression elimination, global copy propagation, and so on. In the optimisation phase of a compiler, the global optimising transformations usually follow the local transformations. The locally optimised code is taken as the input for global optimisation.

Let's take the locally optimised TAC shown in Table 7.26 as the input and analyse the TAC of the entire procedure (spanning across multiple blocks) for opportunities to optimise. The statement (1) in block B0 computes the common sub-expression 2*a, which is used in statement (3) of the same basic block. In block B2, the same common sub-expression '2*a' is computed in statement (7) and used in statement (9) and (12). If we analyse across blocks, we can figure out that (a) the identifier 'a' is not modified between the two computations at (1) and (7); (b) the sub-expression 2*a computed at statement (1) in block B0 can be used in statements (9); and (12) of block B2. By using the value of 2*a computed in (1) throughout the function (at 3, 9 and 12), we can eliminate the re-computation of the common sub-expression '2*a' at (7). This kind of optimising transformations that require analysis and changes spanning across the blocks falls under the purview of global optimisation. Table 7.26 shows the resulting intermediate code after performing the global common sub-expression (2**a*) elimination as discussed above. Observe that the global optimisation resulted in changes in the intermediate code belonging to block B0 and B2.

Table 7.26 *Global optimisation*

#	Locally optimised code		Globally optimised code
B0	(0) proc_begin func (1) x := 2 * a (2) _t2 := 5 * b (3) y := x + _t2 (4) if a > 1 goto .L0		(0) proc_begin func (1) _t13 := 2 * a (2) x := _t13 (3) _t2 := 5 * b (4) y := x + _t2 (5) if a > 1 goto .L0
B1	(5) goto .L1		(6) goto .L1
B2	(6) label .L0 (7) _t4 := 2 * a (8) _t5 := 3 * b (9) _t6 := _t4 + _t5 (10) x := _t6 + 20 (11) _t9 := 4 * b (12) _t10 := _t4 + _t9 (13) y := _t10 + 40	-->	(7) label .L0 (8) _t4 := _t13 (9) _t5 := 3 * b (10) _t6 := _t4 + _t5 (11) x := _t6 + 20 (12) _t9 := 4 * b (13) _t10 := _t4 + _t9 (14) y := _t10 + 40
B3	(14) label .L1 (15) z := x * y (16) return z		(15) label .L1 (16) z := x * y (17) return z
B4	(17) goto .L2		(18) goto .L2
B5	(18) label .L2 (19) proc_end func		(19) label .L2 (20) proc_end func

We have seen how a common sub-expression (2**a*) was eliminated in the global optimisation. We refer to that as global common sub-expression elimination (gcse) indicative of the fact that the common sub-expression was eliminated in a global manner across basic blocks. A common sub-expression eliminated locally within a basic block is termed as local common sub-expression elimination or simply common sub-expression elimination. In a similar manner we use the term *global copy propagation (gcp)* for a copy propagation spanning multiple blocks as opposed to local copy propagation for the ones local to the basic block. The algorithms used for performing global common sub-expression elimination/global copy propagation are different from the ones used in local common sub-expression elimination/local copy propagation. We study about the algorithms performing local common sub-expression elimination and local copy propagation in the section on local optimisation (Section 7.2.9). The algorithms for global common sub-expression elimination and global copy propagation are studied in the section on global optimisation (Section 7.2.10).

The amount of analysis required for local optimisation is lesser, since it is restricted to one basic block at a time. Comparatively the global optimisation requires more analysis, since it requires to analyse intermediate code for the entire function. Due to the lesser analysis involved, the time taken to perform local optimisation on the intermediate code is lesser compared to global optimisation. Another aspect on which local and global optimisation differ is the amount of optimisation feasible. The amount of optimisation possible with local optimisation is lesser than the global optimisation. For example, loop-related optimisations are not possible in local optimisation. This is due to the fact that loop-related optimisations involve analysis of all the blocks involved in a loop at the same time. Since local optimisation concerns analysis and changes localised to a basic block, it is not possible to perform any of the loop-related optimisations in local optimisation. Global transformations are essential for loop optimisation.

Both local and global optimising transformations are performed in most of the compilers. Commonly, the local optimising transformations are performed first, followed by global optimising transformations. Since the local optimisation takes very less time, it is usually included in the preparatory phase for global optimisation.

In one of the studies conducted on the effectiveness of optimisation, Knuth reported an improvement in the speed of execution of about > = 1.4 times in local optimisation and > = 2.7 times due to global optimisation.

In Section 7.2.9, we discuss about local optimisation in detail. We study about how we can implement transformations like local common sub-expression elimination, local copy propagation, dead code elimination, etc. locally as a part of local optimisation in that section.

In Section 7.2.10, we discuss about global optimisation in detail. In that section we learn about implementing global common sub-expression elimination, global copy propagation and other loop-related optimisations.

7.2.9 Local Optimisation

In this section, we learn about implementing local optimisation in a compiler. The optimising transformations that would be studied are local common sub-expression elimination, constant folding, local copy propagation and dead code elimination. We study about algorithms that identify opportunities for the above-mentioned transformations within a basic block and affect the transform on the intermediate code accordingly.

We use the ideas of basic blocks and the directed acyclic graph in implementing the local optimisation.

The local optimisation on the intermediate code is carried out in the steps illustrated in Fig. 7.3.

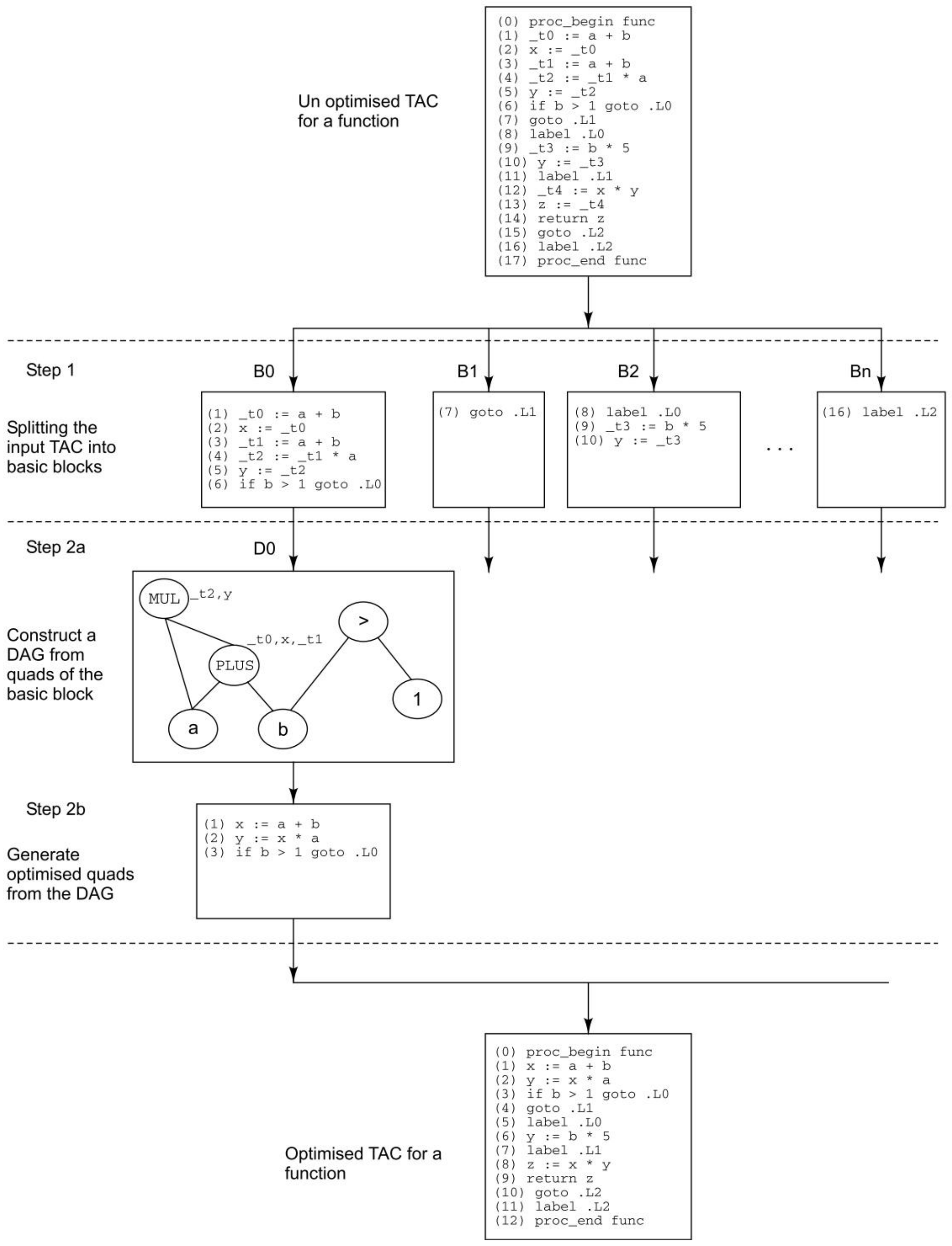
(1) The first step involves splitting the input intermediate code of a function into basic blocks named as B0, B1, B2, and so on till B_n. Figure 7.3 illustrates the split of the input intermediate code into basic blocks B0, through B_n.

(2) For each basic block B in {B0, B1 … B_n}, the following steps are performed:

   (a) A directed acyclic graph D is constructed for the quads in the basic block B using an algorithm described later in this section. Figure 7.3 illustrates the creation of DAG for the block B0.

   (b) The optimised quads are generated from the directed acyclic graph D. Figure 7.3 illustrates the generation of optimised quads from DAG corresponding to the block B0.

   The optimised code for the function is concatenation of the quads generated in 2(b) for all the basic blocks B0 through B_n.


Un optimised TAC for a function
Step 1
B0
B1
B2
Bn
Splitting the input TAC into basic blocks
. . .
Step 2a
D0
Construct a DAG from quads of the basic block
MUL
_t2,y
_t0,x,_t1
PLUS
>
1
a
b
Step 2b
Generate optimised quads from the DAG
Optimised TAC for a function

```
(0) proc_begin func
(1) _t0 := a + b
(2) x := _t0
(3) _t1 := a + b
(4) _t2 := _t1 * a
(5) y := _t2
(6) if b > 1 goto .L0
(7) goto .L1
(8) label .L0
(9) _t3 := b * 5
(10) y := _t3
(11) label .L1
(12) _t4 := x * y
(13) z := _t4
(14) return z
(15) goto .L2
(16) label .L2
(17) proc_end func
```

```
(1) _t0 := a + b
(2) x := _t0
(3) _t1 := a + b
(4) _t2 := _t1 * a
(5) y := _t2
(6) if b > 1 goto .L0
```

```
(7) goto .L1
```

```
(8) label .L0
(9) _t3 := b * 5
(10) y := _t3
```

```
(16) label .L2
```

```
(1) x := a + b
(2) y := x * a
(3) if b > 1 goto .L0
```

```
(0) proc_begin func
(1) x := a + b
(2) y := x * a
(3) if b > 1 goto .L0
(4) goto .L1
(5) label .L0
(6) y := b * 5
(7) label .L1
(8) z := x * y
(9) return z
(10) goto .L2
(11) label .L2
(12) proc_end func
```

Fig. 7.3 *Local optimisation of intermediate code*

We study about each of these steps in detail in the next few sections. Section 7.2.9.1 describes the step 1 followed by an example. Section 7.2.9.3 describes the step 2(a), followed by an example. Section 7.2.9.5 details the step 2(b) followed by an example.

7.2.9.1 Step 1—Splitting the IC into Basic Blocks

As already mentioned, in order to improve the intermediate code, the three address statements are broken up into smaller units on which transformations can be easily applied. Each of these units is called a ***basic block***. A basic block is a sequence of three address statements in which the flow of control enters the beginning and leaves only at the end without any possibility of branching except at the end. In other words, once the control enters a basic block, each and every statement in it is executed.

Let's take an example to understand the idea of splitting up the intermediate into basic blocks. Consider the input C source and its corresponding intermediate code shown in Table 7.27.

Table 7.27 *Input source and its intermediate code*

Input source	**TAC**

`int largest(int p,int q, int r)`
`{`
`    int tmp;`
 	
`    if(p > q){`
`        tmp=p;`
`    }else{`
`        tmp=q;`
`    }`
`    if(r > tmp){`
`        tmp=r;`
`    }`
 	
`    return(tmp);`
`}`
	`(0) proc_begin largest`
	`(1) if p > q goto .L0`
	`(2) goto .L1`
	`(3) label .L0`
	`(4) tmp := p`
	`(5) goto .L2`
	`(6) label .L1`
	`(7) tmp := q`
	`(8) label .L2`
	`(9) if r > tmp goto .L3`
	`(10) goto .L4`
	`(11) label .L3`
	`(12) tmp := r`
	`(13) label .L4`
	`(14) return tmp`
	`(15) goto .L5`
	`(16) label .L5`
	`(17) proc_end largest`

Let's go over the intermediate code shown in Table 7.27 and break it up into basic blocks. By definition, a basic block consists of statements in which the flow of control has to be sequential. The TAC statements 0 and 1 have sequential flow of control and hence form the block 0. The statement 1 is a conditional goto statement, which would transfer the control to label .L0 in case the condition $p > q$ is true. The statement 2 would not be reached when the condition $p > q$ is true. The block B0 ends with statement 1 because there is no assurance that the statement 2 would be reached after statement 1. In general, the statement following a conditional or unconditional goto is a leader or the first statement of the next block. The statement 2 falls in new block B1. The statement 3—label statement, would never be reached sequentially after 2, since statement 2 is a goto statement. Hence statement 3 falls into the next block B2. The statements 3, 4 and 5 have sequential flow of control and hence would go in the same block—B2. The flow of control would not reach statement 6 after statement 5 and hence the block B2 ends with statement 5. The statements 6 and 7 have sequential flow of control and go into block B3. The statement 8, which is a label statement, cannot be in the same block because control can directly come to label from a conditional or unconditional goto statement. Since there is no assurance that the control flow would follow statement 8 after 7, the statement 8 goes into the new block B4. In general, a label statement is a leader or the first statement of a new block, since control can come to it from a goto statement or sequentially. The reader is advised to apply the above-mentioned principles and understand the rationale behind the breaking of statements into blocks B5 through B8. Figure 7.4 shows the complete split of the intermediate code into basic blocks.

The method we used above to partition the TAC statements into basic blocks is formalised in Algorithm 7.1. We use the idea of leaders, the first statements in the IC of basic blocks to make the partitions. The leaders are (a) Label statement and (b) Any statement, following a conditional or unconditional goto statement. The statements 0, 2, 3, 6, 8, 10, 11, 13 and 15 in the above example are all leaders because they satisfy at least one of the criteria mentioned above. Each block consists of the leader and all the statements following it up to the next leader, but not including it.

```
(0) proc_begin largest        B0
(1) if p > q goto .L0
-----------------------------
(2) goto .L1                  B1
-----------------------------
(3) label .L0                 B2
(4) tmp := p
(5) goto .L2
-----------------------------
(6) label .L1                 B3
(7) tmp := q
-----------------------------
(8) label .L2                 B4
(9) if r > tmp goto .L3
-----------------------------
(10) goto .L4                 B5
-----------------------------
(11) label .L3                B6
(12) tmp := r
-----------------------------
(13) label .L4                B7
(14) return tmp
-----------------------------
(15) label .L5                B8
(16) proc_end largest
```

Fig. 7.4 *Splitting of IC into basic blocks*

```
split_into_basic_blocks()
{
        i=0
        prev=NULL

        /* L is a list of TAC statements */
        /* B is a data structure associating quads with a basic block */

        Add the first statement of L into Bi
        for every statement s in L
        do
        {
             leader=0
             if(s is a label statement){
                    leader=1
             }
             if(prev is a conditional or unconditional goto){
                    leader=1
             }
             if(leader == 1){
                    i=i+1;
             }
             Attach s to Bi
             prev=s
        }
}
```

Algorithm 7.1 *Partition TAC statements into basic blocks*

7.2.9.2 *Example 1—Splitting TAC Statements to Basic Blocks* This section demonstrates the toy C compiler (mycc) splitting the intermediate code into basic blocks using Algorithm 7.1 described in the preceding section. The toy C compiler takes as input, a sample C input source and gives out (a) TAC and (b) the breakup of the TAC into basic blocks. The dialog below shows 'mycc' taking in some sample input C sources, and printing out their intermediate code in TAC format along with the basic block information.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g ++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g ++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$  g  ++ -DCHAP7_EX1  -DICGEN  -g  -Wall  ic_gen.cc  optimize.cc  target_code_gen.cc  mycc.cc
semantic_analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# An input source
$ cat -n test1.c
    1  int largest(int p, int q, int r)
    2  {
    3       int tmp;
    4
    5       if(p > q){
    6            tmp=p;
    7       }else{
    8            tmp=q;
    9       }
   10       if(r > tmp){
   11            tmp=r;
   12       }
   13
   14       return(tmp);
   15  }

# Partitioning the IC into Basic Blocks
$ ./mycc -i -O local -v test1.c
TAC Before optimisation
(0) proc_begin largest
(1) if p > q goto .L0
(2) goto .L1
(3) label .L0
(4) tmp: = p
(5) goto .L2
(6) label .L1
(7) tmp: = q
(8) label .L2
(9) if r > tmp goto .L3
```

```
(10) goto .L4
(11) label .L3
(12) tmp: = r
(13) label .L4
(14) return tmp
(15) goto .L5
(16) label .L5
(17) proc_end largest

Quads After Splitting into Basic Blocks
BLOCK =0
(0) proc_begin largest
(1) if p > q goto .L0
BLOCK =1
(2) goto .L1
BLOCK =2
(3) label .L0
(4) tmp := p
(5) goto .L2
BLOCK =3
(6) label .L1
(7) tmp := q
BLOCK =4
(8) label .L2
(9) if r > tmp goto .L3
BLOCK =5
(10) goto .L4
BLOCK =6
(11) label .L3
(12) tmp := r
BLOCK =7
(13) label .L4
(14) return tmp
(15) goto .L5
BLOCK =8
(16) label .L5
(17) proc_end largest

# Another input source
$ cat -n test1a.c
    1  int fact(int num)
    2  {
    3       int i,prod;
    4
    5       i=1;
    6       prod=1;
    7
    8       while(i<=num){
    9            prod=prod*i;
   10            i=i + 1;
   11       }
   12
   13       return(prod);
   14  }

# Basic Blocks for the input source
```

```
$ ./mycc -i -O local -v test1a.c
TAC Before optimisation
(0) proc_begin fact
(1) i := 1
(2) prod := 1
(3) label .L0
(4) if i <= num goto .L1
(5) goto .L2
(6) label .L1
(7) _t0: = prod * i
(8) prod: = _t0
(9) _t1: = i + 1
(10) i: = _t1
(11) goto .L0
(12) label .L2
(13) return prod
(14) goto .L3
(15) label .L3
(16) proc_end fact

Quads after Splitting into Basic Blocks
BLOCK =0
(0) proc_begin fact
(1) i := 1
(2) prod := 1
BLOCK =1
(3) label .L0
(4) if i <= num goto .L1
BLOCK =2
(5) goto .L2
BLOCK =3
(6) label .L1
(7) _t0 := prod * i
(8) prod := _t0
(9) _t1 := i + 1
(10) i := _t1
(11) goto .L0
BLOCK =4
(12) label .L2
(13) return prod
(14) goto .L3
BLOCK =5
(15) label .L3
(16) proc_end fact
```

7.2.9.3 Step 2(a)—Construction of DAG from Basic Block

In this section, we study about how we can construct a directed acyclic graph (DAG) from the quads in a basic block. As introduced before, a DAG is a data structure used for implementing optimising transformations on the intermediate code within a basic block. Each basic block is transformed into a DAG.

We have seen earlier that a DAG Node consists of a label and an optional list of attached identifiers. We can represent a DAG node by a structure or a class containing the following main elements:

label	:	The label for a leaf node is the identifier or the constant that the node represents. For the interior nodes, it is the operator symbol represented by the node.
attached_identifiers_list	:	This is a list of identifiers attached to the node. The current value of each of the identifiers in the list is represented by the current node.

The process of building a DAG requires an association between an identifier and a DAG node. The DAG node represents the 'current' value of the Identifier. A DAG node is associated with an identifier by using the function set_current(identifier, dag_node). We can fetch the DAG node associated with an identifier by using the function get_current(identifier).

set_current (identifier,dag_node)	:	This associates the current value of an identifier with the DAG node.
get_current (identifier)	:	This returns the dag node associated with the identifier. This represents the current value of identifier.

During the DAG construction process, there are functions needed for adding and removing an identifier from the attached list of identifiers of a DAG node. The following functions are used for the purpose.

add_attached_identifier(dag_node,identifier)	:	This function adds the identifier to the list of attached identifiers of the DAG node.
remove_attached_identifier(dag_node,identifier)	:	This function removes the identifier from the attached list of identifiers of the DAG node.

In the algorithm presented a little later for constructing a DAG from the TAC statements, the TAC statements are represented using a generic form res: = arg1 OP arg2. The operator OP determines which of the three operands are defined. For example, an ADD operator would have all the three operands res, arg1 and arg2 as defined. A unary minus operator (UMINUS) would have the operands res and arg1 as defined but the operand arg2 is undefined. In a similar way, the ASSIGN operator has res and arg1 as valid, but arg2 as undefined.

We classify the three address operators into 2 classes based on the ability to participate in common sub-expression identification and elimination.

The TAC operators belonging to class 1 are the ones for which the result 'res' is valid and can be used for identifying the common sub-expressions. Examples of these are quads using the operators ADD, MUL, DIV, and so on.

The TAC operators in class 2 are the ones that cannot participate in the common sub-expression identification and elimination either due to (a) 'res' cannot be considered as a 'result' of an expression, e.g. LT,GT, L_INDEX_ASSIGN or (b) 'res' is not valid, e.g. PROC_BEGIN,PARAM, etc.

Table 7.28 shows the classification of all the 23 operators defined in our intermediate language that we saw in Chapter 5.

Table 7.28 *Classification of TAC operators*

#	TAC operator	Description	Class
1	ADD, MUL, DIV, SUB, UMINUS, ADDR_OF, ASSIGN, R_INDEX_ASSIGN	res is valid and can be used as a result for identifying the common sub-expressions.	Class 1

2	LT, GT, LE, GE, EQ, NE, L_INDEX_ASSIGN	arg1, arg2 and res are valid, but we cannot use 'res' as a result for identifying the common sub-expressions. The quads with these operators cannot participate in common sub-expression identification.	Class 2
3	PROC_BEGIN, PROC_END, RETURN, RETRIEVE, PARAM, CALL, LBL, GOTO	arg2 and res are not valid. Only arg1 is valid. The quads with these operators cannot participate in common sub-expression identification.	Class 2

We now turn to the algorithm for creating a DAG from the quads in a basic block. We use the functions that have been mentioned above. The DAG is represented by the structure described previously.

For each TAC statement res: = arg1 OP arg2 in the basic block do steps 1 to 3:

1. If arg1 is valid, then find a dag node 'node_arg1' that represents the current value of 'arg1'. If we do not find a node, create a node and call it 'node_arg1'. Now, the 'node_arg1' represents the current value of 'arg1'.
2. If 'arg2' is valid, then find a dag node 'node_arg2' that represents the current value of 'arg2'. If we do not find a node, create a node and call it 'node_arg2'. Now the 'node_arg2' represents the current value of 'arg2'.
3. If the OP is ASSIGN operator, then,

   if 'res' was attached to some other dag node previously, remove that linkage. Associate the current value of identifier 'res' with 'node_arg1'. Add the identifier 'res' to the list of identifiers attached to 'node_arg1'.

   Else, if the OP is one of class 1 operators, i.e. operators on which we can have a common sub-expression then,

   Find if there is an interior node 'node_res', whose children are 'node_arg1' and 'node_arg2'. If we do not find such a node then create a new dag node called 'node_res' and make 'node_arg1' and 'node_arg2' as its children. In either case, i.e. 'node_res' was created freshly now, or found already, do the following (a) if res were attached to some other dag node previously, remove that linkage. Associate the current value of identifier 'res' with 'node_res' (b) add the identifier 'res' to the list of identifiers attached to 'node_res'.

   Else,

   /* class 2 operators */

   Find if there is a node 'node_res' that represents the current value of 'res'. If there is no node representing the current value of res, then create a new dag node 'node_res' and associate the current value of identifier 'res' with 'node_res'. Create a new dag node 'n4', whose children are node_res, node_arg1, node_arg2.

Algorithm 7.2 shows the above in a loose C pseudo-code form. The repeated tasks in the algorithm have been converted to functions. The three support functions, namely get_dag_node(), move_attached_identifier() and find_dag_node are also presented in Algorithm 7.2.

```
construct_dag ()
{

      for (each 'quad' in the Basic Block) {
           arg1  = quad.arg1 ;
           arg2  = quad.arg2 ;
           res   = quad.res ;
           op   = quad.op ; /* Operator of the quad */

           /* Step 1 */
           node_arg1  = get_dag_node(arg1)

           /* Step 2 */
           node_arg2  = get_dag_node(arg2)

           /* Step 3 */
           if (op is ASSIGN){

                move_attached_identifier(res,node_arg1);

           } else if (op is ADD or MUL or DIV or SUB or UMINUS or
                     ADDR_OF or R_INDEX_ASSIGN){/* Class 1*/

                node_res  = find_dag_node(op,node_arg1,node_arg2);
                if(node_res = = NULL){
                     node_res  = mk_dag_node(op,node_arg1,node_arg2);
                }

                move_attached_identifier(res,node_res);

           }else{
                /*
                    Class 2 Operators
                    op is one of
                    LT,GT,LE,GE,EQ,NE,L_INDEX_ASSIGN - 3 args
                    PROC_BEGIN,PROC_END,RETURN,RETRIEVE,PARAM,CALL,LBL,GOTO - 1 arg

                */

                node_res  = get_dag_node(res)

                n4  = mk_dag_node(op,node_arg1,node_arg2,node_res);
           }
      }
}

dag_node *get_dag_node(sym_tab_entry *s)
{
    dag_node *d;

    if(s == NULL){
         return(NULL);
    }

    if((d=get_current(s)) == NULL) {

         /* create a leaf node */
         d = mk_dag_node (s);

         /*
              dag_arr is a global data structure containing pointers to all
```

```
                the dag nodes created
           */
           dag_arr[dag_node_no]=d;
           dag_node_no = dag_node_no  + 1;

           /* set the associated node */
                set_current(s,d);
     }

     return(d);
}

move_attached_identifier(sym_tab_entry *s,dag_node *d)
{
   dag_node *tmp;

   /* Remove the existing linkage */
   tmp  = get_current(s);
   if(tmp ! = NULL){
      remove_attached_identifier(s,tmp);
   }

   /* Set the new association */
   set_current(s,d);

   /* Attach the identifier to the dag node */
   add_attached_identifier(s,d);
}

dag_node * find_dag_node(op, dag_node *l,dag_node *r)
{
   dag_node *d;

   /*
   Search in dag_arr to see, if there is a dag node with op as label,
   with 'l' as the left child and 'r' as the right child
   */

   n  = dag_node_no;
   for(i=0;i<n;i++)
   {
       d=dag_arr[dag_node_no];

       if(
           (d->label == op)  &&
           (d ->left_child == l)  &&
           (d ->right_child == r)

          ){

               return(d);
      }
   }
   return(NULL);
}
```

Algorithm 7.2 *Construction of DAG*

7.2.9.4 Illustration of Construction of a DAG Consider the input C source file and the corresponding unoptimised intermediate code (see Table 7.29) for demonstrating the construction of DAG using Algorithm 7.2. The intermediate code can be divided into 2 basic blocks. The basic block 0 consists of statements (1) through (12) with both inclusive. Block 1 consists of the TAC statements (13) and (14).

Table 7.29 *Input source and intermediate code*

Input source	TAC

`int a,b,c,d,e,f,g;`
 	
`void func()`
`{`
`      a  = (b  + c)*d ;`
`      e  =  f * a ;`
`      f  = (b + c)*e;`
`      g  = (b + c)/d;`
`}`
	`(1) proc_begin func`
	`(2) _t0: = b + c`
	`(3) _t1: = _t0 * d`
	`(4) a: = _t1`
	`(5) _t2: = f * a`
	`(6) e := _t2`
	`(7) _t3 := b + c`
	`(8) _t4 := _t3 * e`
	`(9) f := _t4`
	`(10) _t5: = b + c`
	`(11) _t6: = _t5 / d`
	`(12) g := _t6`
	`(13) label .L0`
	`(14) proc_end func`

We will build the DAG for block 0 in a step-by-step fashion using Algorithm 7.2. The intermediate code of block 0 is shown in Fig. 7.5.

The processing of TAC statement (1) proc_begin func causes the creation of a leaf for the identifier ‘func’ and an interior node with PROC_BEGIN as the operator as shown in Fig. 7.6.

```
(1) proc_begin func
(2) _t0 := b + c
(3) _t1 := _t0 * d
(4) a := _t1
(5) _t2 := f * a
(6) e := _t2
(7) _t3 := b + c
(8) _t4 := _t3 * e
(9) f := _t4
(10) _t5 := b + c
(11) _t6 := _t5/d
(12) g := _t6
```

Fig. 7.5 *Basic block 0*

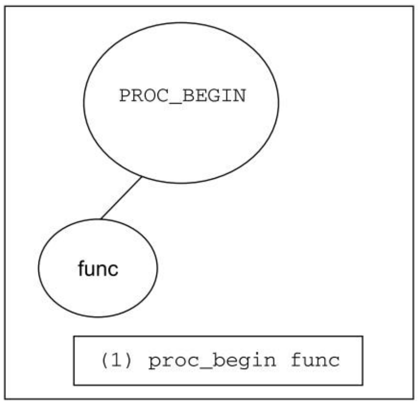


Fig. 7.6 *Processing of (1) PROC_BEGIN func*

The TAC statement (2) _*t*0: = *b* + *c* causes the creation of a leaf for the identifier ‘*b*’ and another leaf for the identifier ‘*c*’. An interior node for the operator ADD is created with the leaf nodes of ‘*b*’ and ‘*c*’ as children. The identifier ‘_*t*0’ is attached to the created interior node as shown in Fig. 7.7.

The next TAC statement to be processed is (3) _*t*1: = _*t*0 * *d*. This uses the existing node to which ‘_*t*0’ is attached. A new leaf node is created for the identifier ‘*d*’. An interior node with a label ‘MUL’ is created.

The identifier _t1 is attached to this interior node. The DAG at this point is shown in Fig. 7.8. The dotted line in Fig. 7.8 shows the part of the DAG that has been created due to the processing of the current TAC statement (3) _*t*1: = _*t*0 * *d*.

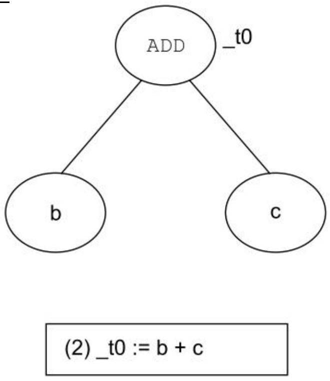


Fig. 7.7 *Processing (2) _t0: = b + c*

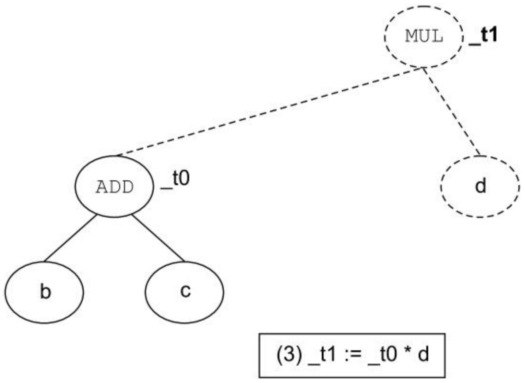


Fig. 7.8 *DAG after processing (3) _t1: = _t0 * d*

The next TAC statement to be processed is the assignment statement (4) *a*: = _*t*1. This attaches the identifier '*a*' to the DAG node associated with the current value of the identifier '_*t*1'. The DAG at this point is shown in Fig. 7.9. The attaching of the identifier '*a*' to the DAG node holding the current value of '_*t*1' is shown in bold for the sake of clarity.

The processing of next TAC statement (5) _*t*2: = *f* * *a*, causes the creation of a new leaf node storing the value of the identifier '*f*', since there is no DAG node associated with '*f*'. An interior node with the label MUL is created with the children as DAG nodes storing the latest value of identifiers '*f*' and '*a*'. Figure 7.10 shows the DAG after the processing of (5) _*t*2 := *f* * *a*. The dotted line shows the part of the DAG created due to the processing of (5) _*t*2: = *f* * *a*.

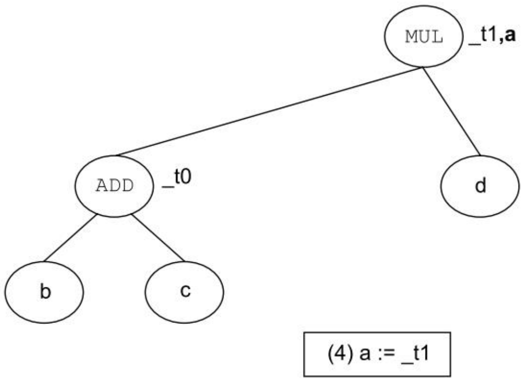


Fig. 7.9 *DAG after processing (4) a := _t1*

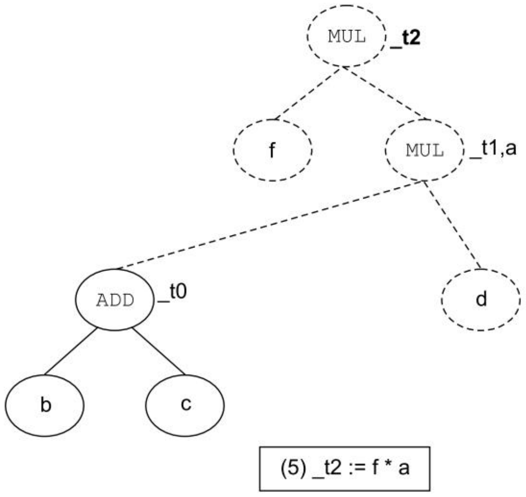


Fig. 7.10 *DAG after processing (5) _t2 := f * a*

The next statement to be processed is the assignment statement (6) *e* := _*t*2. TAC This causes an additional identifier '*e*' to be attached to the DAG node storing the latest value of '_*t*1' as shown in Fig. 7.11. Observe that any DAG node with multiple identifiers attached to it indicates that all of those identifiers contain the same value at that point in time.

The next TAC statement to be processed is (7) *_t*3 := *b* + *c*. We find that there exists an interior node ADD that has '*b*' and '*c*' as children. We attach the identifier '*_t*3' to such a node as shown in Fig. 7.12. From the DAG shown in Fig. 7.12, we can observe that the identifiers '*_t*0' and '*_t*3' hold the value of a common expression '*b* + *c*'. We can also notice that the identifiers '*_t*1' and '*a*' hold the value of common expression (*b* + *c*) * *d*.

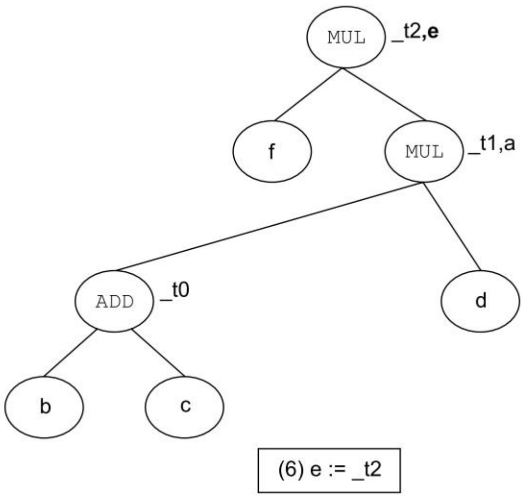


Fig 7.11 *DAG after processing (6) e := _t2*

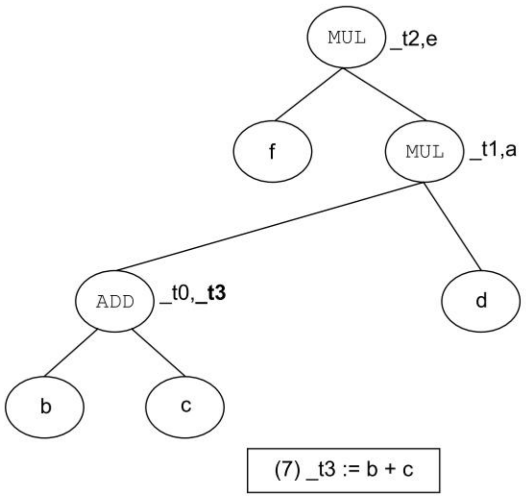


Fig. 7.12 *DAG after processing (7) _t3 := b + c*

The next TAC statement to be processed is (8) *_t*4: = *_t*3 * *e*. An interior node with the label MUL is created with the children as DAG nodes storing the latest value of identifiers '*_t*3' and '*e*'. We attach the identifier '*_t*4' to such a node as shown in Fig. 7.13. The dotted line shows the part of the DAG created due to the processing of (8) *_t*4: = *_t*3 * *e*.

The next TAC statement to be processed is the assignment statement (9) *f*: = *_t*4. This causes an additional identifier '*f*' to be attached to the DAG node storing the latest value of '*_t*4' as shown in Fig. 7.14.

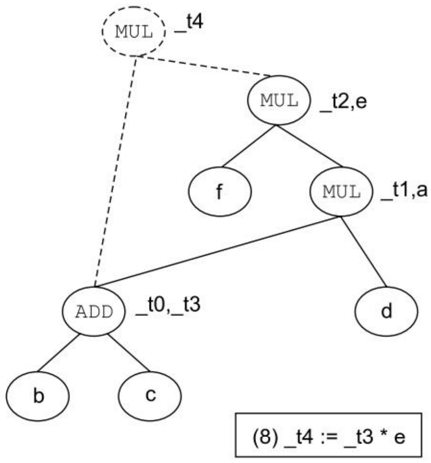


Fig. 7.13 *DAG after processing (8) _t4 := _t3 * e*

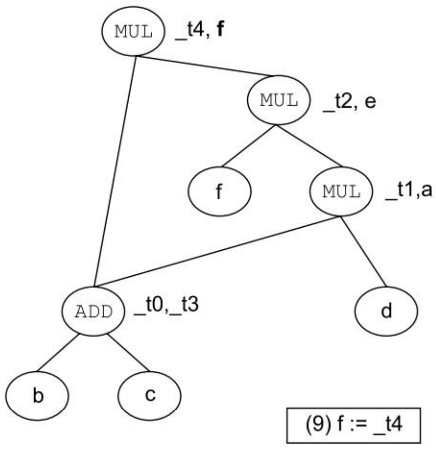


Fig. 7.14 *DAG after processing (9) f := _t4*

The next TAC statement to be processed (10) *_t5* := *b* + *c*. We find that there exists an interior node ADD that has '*b*' and '*c*' as children. We attach the identifier '*_t5*' to such a node as shown in Fig. 7.15. From the DAG shown in Fig. 7.15, we can observe that the identifiers '*_t0*', '*_t3*' and now '*_t5*' hold the value of the common expression '*b* + *c*'.

The next TAC statement to be processed (11) *_t6* := *d* / *_t5*. There exists no interior node DIV that has '*d*' and '*_t5*' as children. We create a new interior node DIV having the nodes holding the latest value of '*d*' and '*_t5*' as children as shown by the dotted line in Fig. 7.16. The interior node DIV created newly is attached with the identifier '*_t6*'. The identifier '*_t6*' was not attached to any other node previously, so there is no previous linkage to be discarded.

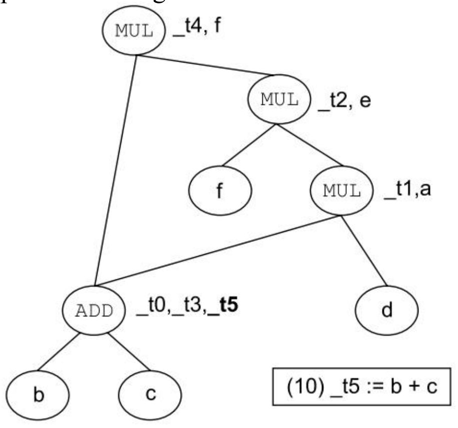


Fig. 7.15 *DAG after processing (10) _t5 := b + c*

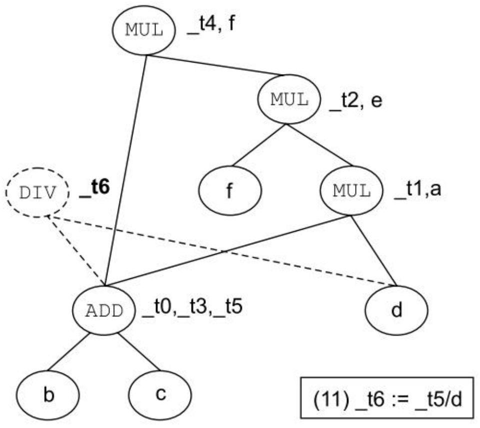


Fig. 7.16 *DAG after processing (11) _t6 := _t5/d*

The next TAC statement to be processed is the assignment statement (12) *g*: = *_t6*. This causes an additional identifier '*g*' to be attached to the DAG node storing the latest value of '*_t6*' as shown in Fig. 7.17. The identifier '*g*' was not attached to any other node previously, so there is no previous linkage to be discarded.

We have completed the processing of all the TAC statements in the basic block 0. The final DAG for the basic block 0 is shown in Fig. 7.18.

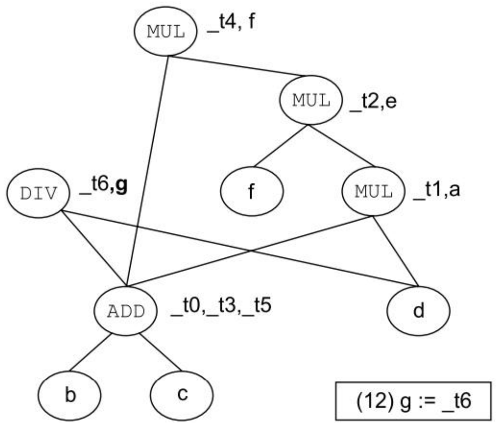


Fig. 7.17 *DAG after processing (12) g := _t6*

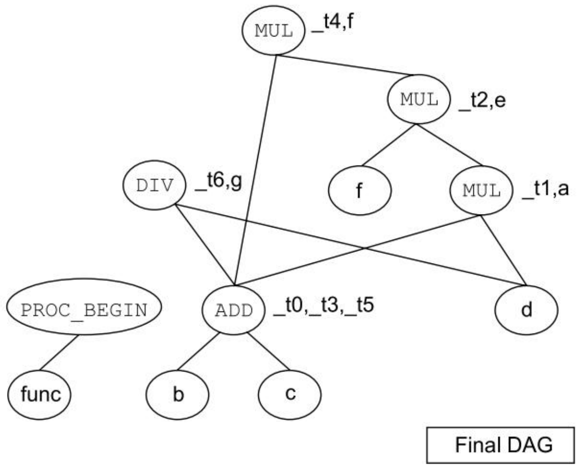


Fig. 7.18 *The final DAG for basic block 0*

7.2.9.5 Step 2(b)—Reconstruction of Intermediate Code from DAG In this section, we study about traversing the DAG to generate optimised intermediate code. The reconstruction of the intermediate code from the DAG is the final step in the local optimisation as seen in Fig. 7.3.

The reconstruction of TAC from the DAG is accomplished by traversing the nodes of a DAG in a topologically sorted order and generating code. In topological sorted order, an interior node is not visited unless all its children have already been visited. The order in which DAG nodes are created from the intermediate code using Algorithm 7.2 presented earlier is itself in a topologically sorted order. Fig. 7.19 shows the DAG created in Section 7.2.9.4, with the nodes numbered 1, 2, 3 and so on in the order they were created. Observe that in the DAG in Fig. 7.19, the nodes marked 1, 2, 3 and so on, are in a topologically sorted order with the parent nodes being created only after all of their children have been created.

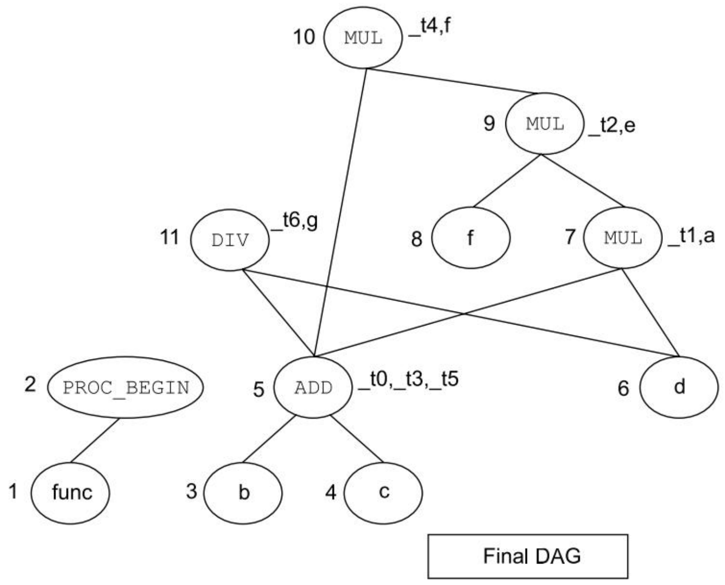


Fig. 7.19 *The order of creation of DAG nodes*

The optimised TAC is generated from the DAG by visiting all the DAG nodes in a topologically sorted order. For each node in the DAG in the topologically sorted order, we do the following:

If the node is a leaf node, then check if it has any attached identifiers. If there are no attached identifiers, no quads are generated for the node. If there are attached identifiers then generate ASSIGN statements that assign the value of the node to each one of the user defined identifiers 's' in the attached identifiers list. Figure 7.20 shows the re-constructed quads from a leaf node in a DAG.

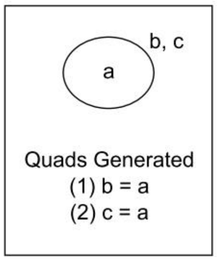


Fig. 7.20 *Quad generation for a leaf node*

If the node is an interior node then we check to see, if the operator is a class 1 operator or a class 2 operator. As explained before, class 1 operators have a 'result' amenable to common sub-expression elimination.

If it is class 2 operator node, generate a quad with the operator, left child, right child and any other additional children if present. Figure 7.21 shows the quads generated for an interior node using PROC_BEGIN—a class 2 operator.

If it is a class 1 operator node, then generate a quad that assigns the value of the node to one of the identifiers '*s*' in the attached identifiers list. The preference is given to an identifier '*s*' in the list of identifiers attached to the node, if it is a user-defined variable as opposed to a compiler generator temporary. The value of the node is produced by applying the operator on the left and right child. Figure 7.22 shows the quads generated from an interior node having children that are leaves. In Fig. 7.22, the identifier '*d*' has been selected for assigning the value, since it is user defined as compared to _*t*0. The identifier '*d*' now represents this node for all further computations. In case there were other user defined identifiers like say *e* and *f* also attached to this interior node, additional assignment statements $e = d, f = d$ are generated at this point.

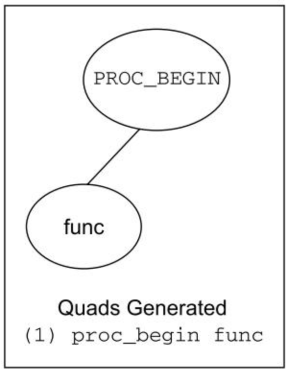


Fig. 7.21 *Quad generation for interior node (class 2 operator)*

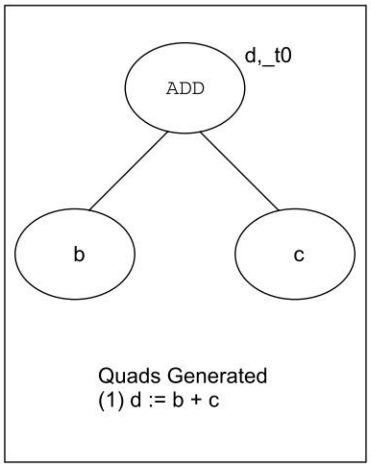


Fig. 7.22 *Quad generation for interior node having leaves as children*

The generation of quads for interior nodes having other interior nodes as children is also similar. Figure 7.23 shows an interior DAG node (shaded in gray) having other interior nodes as children. The quad generated in this case would be $m = j + k$, since '*j*' and '*k*' are selected identifiers for child interior nodes. By virtue of the fact that the nodes are visited in a topographical order, each of the children (interior nodes) would already have a selected identifier representing the node. In Fig. 7.23, we have the left child being represented by the selected identifier '*j*' and right child by the selected identifier '*k*'.

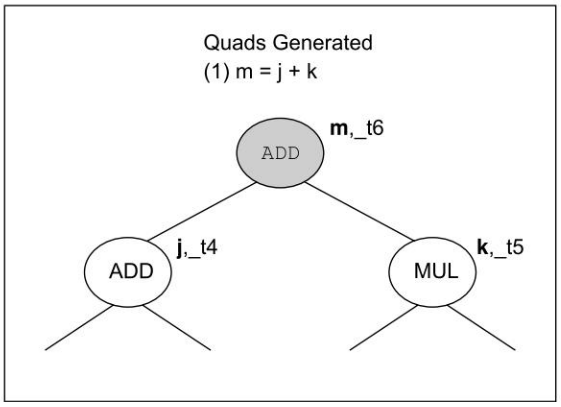


Fig. 7.23 *Reconstruction of quads from interior node having other interior nodes as children*

In situations where there are no user-defined identifiers in the list of identifiers attached to the interior node, we can pick any one of the temporary identifiers to store the value of the node. If there are no identifiers at all in the attached identifiers list, create a new unused temporary variable, let's say '*_tk*' and generate a quad that assigns the value of the node to it.

Algorithm 7.3 formalises the regeneration of optimised quads from the DAG as explained above.

```
dag_arr is the array of all the dag nodes. It is in the same order as created and hence
topographically sorted.

regenarate_quads ()
{
        for (each dag node 'd' in the dag_arr) {

                if(d is a leaf node with label 'l'){
                        if(there are attached identifiers to the node i1,i2 in){
                                Generate assign quads for each of the user defined (non-
                                temporarary) identifiers (i1=l,i2=l etc)
                        continue;
                }

                /* 'd' is an Interior Node */

                if (operator is not one of ADD MUL DIV SUB UMINUS ADDR_OF R_INDEX_ASSIGN){
                        /* These are operators for which 'result' is undefined */
                        Generate a quad with (operator, left child,right child and additional
                        child if any)
                        continue;
                }
                /* These are operators for which 'result' is defined */
                /* operator is one of ADD MUL DIV SUB UMINUS ADDR_OF R_INDEX_ASSIGN */

                Select an user defined identifier 's' in the attached identifier's list.

                Generate a new quad with (operator,left child, right child, s)

                if (There are more than one user defined identifiers in the attached
                identifier's list){
                        Generate assign quads for i1=s,i2=s,i3=s
                }
        }
}
```

Algorithm 7.3 *Regenerating optimised quads from DAG*

7.2.9.6 Illustration of Reconstruction of Intermediate Code from DAG Consider the final DAG for the basic block 0 discussed earlier in 7.2.9.4 and shown again in Fig. 7.24 for understanding the re-construction of quads using Algorithm 7.3.

Table 7.30 shows the regeneration of the quads from the DAG in a step-by-step fashion based on Algorithm 7.3 and the discussion in Section 7.2.9.5. The DAG in Fig. 7.24 showing nodes numbered in the order of creation is used as the basis for regeneration of the quads.

In Table 7.30, the node number is mentioned in the first column. The generated quad for the same node based on Algorithm 7.3 is shown in the second column. The explanation column gives the details with regard to the working of the algorithm.

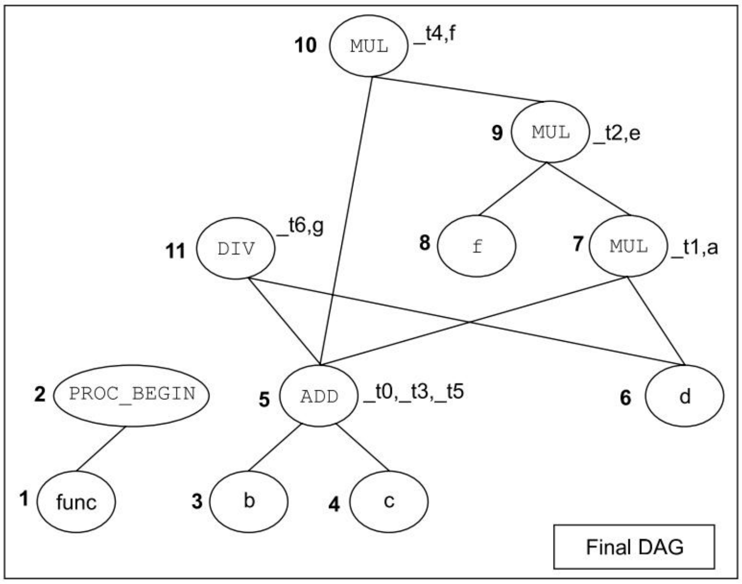


Fig. 7.24 *Final DAG considered for regeneration of optimised quads*

Table 7.30 *Regeneration of optimised quads from DAG*

Node #	Generated Quad	Explanation
1	None	It is leaf node, without any attached identifiers. No Quads are generated.
2	proc_begin func	This node is an interior node. It is a class 2 operator. The Quad is generated with the Left child.
3	None	It is leaf node, without any attached identifiers. No Quads are generated.
4	None	It is leaf node, without any attached identifiers. No Quads are generated.
5	_t0 = b + c	This node is an interior node. The left child is the leaf node ‘b’. The right child is the leaf node ‘c’. The operator of the node is ADD. Hence, the value of the node is (b+c). We select the identifier ‘_t0’ among the attached identifiers _t0,_t3,_t5 for assigning the value of the node. We could have selected any one of the three attached identifiers (_t0,_t3,_t5), we chose _t0. We will continue to use _t0 as the identifier reflecting the value of this node (Node # 5) till the completion of the algorithm.
6	None	It is leaf node, without any attached identifiers. No Quads are generated.
7	a := _t0 * d	This node is an interior node. The left child is node number 5, which had selected the identifier _t0 for saving the value of the node. The right child is the leaf node with label ‘d’. The operator of the node is MUL. Hence, the value of the current node is (_t0 * d). We select the identifier ‘a’ among the attached identifiers _t1,a for assigning the value of the node. The reason for selecting ‘a’ ahead of _t1 is that ‘a’ is a user-defined identifier. In the generated quad, the value of the node _t0 * d, is assigned to the selected identifier ‘a’.
8	None	It is leaf node, without any attached identifiers. No Quads are generated.

9	e = f * a	This node is an interior node. The left child is a leaf node with label 'f'. The right child is node number 7, which had selected the identifier 'a' for assigning the value of the node. The operator of the node is MUL. Hence, the value of the current node is (f * a). We select the identifier 'e' among the attached identifiers _t2,e for assigning the value of the node. The reason for selecting 'e' ahead of _t2 is that 'e' is a user-defined identifier. In the generated quad, the value of the node f * a , is assigned to the selected identifier 'e'.
10	f = _t0 * e	This node is an interior node. The left child is node number 5, which had selected the identifier _t0 for assigning the value of the node. The right child is node number 9, which had selected the identifier 'e' for assigning the value of the node. The operator of the node is MUL. Hence, the value of the current node is (_t0 * e). We select the identifier 'f' among the attached identifiers _t4, f for assigning the value of the node. The reason for selecting 'f' ahead of _t4 is that 'f' is a user-defined identifier. In the generated quad, the value of the node _t0 * e , is assigned to the selected identifier 'f'.
11	g := _t0 / d	This node is an interior node. This left child is node number 5, which had selected the identifier _t0 for assigning the value of the node. The right child is the leaf node with label 'd'. The operator of the node is DIV. Hence, the value of the current node is (_t0 / d). We select the identifier 'g' among the attached identifiers _t6,g for assigning the value of the node. The reason for selecting 'g' ahead of _t6 is that 'g' is a user-defined identifier. In the generated quad, the value of the node _t0 / d , is assigned to the selected identifier 'g'.

The optimised quads regenerated from the DAG are given in Table 7.31. The optimised TAC contains 6 quads. The un-optimised TAC before optimisation containing 12 instructions is also as shown in Table 7.31.

Table 7.31 *Optimised TAC regenerated from the DAG for basic block 0*

Un-optimised code	**Optimised code regenerated from DAG**

`(1) proc_begin func`
`(2) _t0 := b + c`
`(3) _t1 := _t0 * d`
`(4) a := _t1`
`(5) _t2 := f * a`
`(6) e := _t2`
`(7) _t3 := b + c`
`(8) _t4 := _t3 * e`
`(9) f := _t4`
`(10) _t5 := b + c`
`(11) _t6 := _t5 / d`
`(12) g := _t6`
	`(1) proc_begin func`
	`(2) _t0 := b + c`
	`(3) a := _t0 * d`
	`(4) e := f * a`
	`(5) f := _t0 * e`
	`(6) g := _t0 / d`

7.2.9.7 Optimising Transformations on DAG The conversion of the quads pertaining to a basic block into DAG and the subsequent generation of the optimised quads from the DAG results in several optimising transformations taking place on the input quads. In this section, we discuss about how the optimising transformations like common sub-expression elimination, dead store elimination, copy propagation, etc. occur during the process of constructing a DAG and subsequent regeneration of the optimised quads.

In the algorithm for constructing the DAG from the input quads (Algorithm 7.2), there is a check made to find if there is an existing node with the same children (given by the line *node_res* = *find_dag_node* (*op*, *node_arg*1, *node_arg*2);). A new node is created only when such a node does not exist. This action allows us to detect ***common sub-expressions*** and eliminate the re-computation of the same. For example, consider the DAG shown in Fig. 7.24 resulting from the basic block in Fig. 7.5. The common sub-expression (*b* + *c*) is computed thrice in the input code at the quads (2), (7) and (10). These three computations are translated into a single node in the DAG (Node #5) having attached identifiers as _*t*0, _*t*3 and _*t*5. When the reconstruction of the quads takes place from the DAG, the computation is only carried out once, stored in the selected identifier _*t*0 and reused later. This illustrates how the DAG construction scheme identifies the common sub-expressions and helps in eliminating its re-computation later during the reconstruction of the optimised quads from the DAG.

During the DAG construction process multiple identifiers are attached to a DAG node. All of these identifiers hold the same value. During the reconstruction of the quads from the DAG, we select only one of the identifier among the attached identifier list to represent the value. We generate additional assignment statements only if there are other user defined variables in the attached identifier list. This eliminates unnecessary assignments of the form *a*: = *b*. For example, consider the DAG shown in Fig. 7.24 resulting from the basic block in Fig. 7.5. The input quad (5) _*t*2 := *f* * *a* resulted in the DAG node 9. The next input quad (6) *e* := _*t*2, resulted in adding an attached identifier '*e*' to the DAG node #9. When the DAG is traversed for generation of optimised quads, the node 9 results in a single quad *e* := *f* * *d*. No assignment is made to _*t*2, since it is not a user-defined variable. This eliminated the assignment *e* := _*t*2. Also, it allows the use of '*e*' instead of _*t*2 (copy propagation) in further generation of quads. This illustrates how ***copy propagation*** is facilitated in the process of construction of DAG and the subsequent regeneration of optimised quads.

The elimination of common sub-expression and copy propagation transformations are implicit to the process of construction of DAG and the subsequent regeneration of optimised quads. There are other optimising transformations like the constant folding and elimination of dead code that can also be accomplished by revising the DAG before the generation of optimising quads.

Constant folding can be implemented by traversing the DAG and trimming it at the points where there are constants as children. Consider the sample input source and its corresponding intermediate code shown in Table 7.32. The DAG built for block 0 using Algorithm 7.2 is shown in Fig. 7.25. The nodes of the DAG shown in Fig. 7.25 are numbered according to the order of creation. The segment of the DAG containing PROC_BEGIN has been ignored for simplicity.

Table 7.32 *Input source and intermediate code*

Input source	**TAC**

```
1  int a[45];
2
3  int func()
4  {
5
6     a[5]=25;
7     a[6]=30;
8  }
```
	```
	(1) proc_begin func
	(2) _t0 := 5 * 4
	(3) _t1 := &a
	(4) _t1[_t0] := 25
	(5) _t2 := 6 * 4
	(6) _t3 := &a
	(7) _t3[_t2]: = 30
	(8) label .L0
	(9) proc_end func
	```

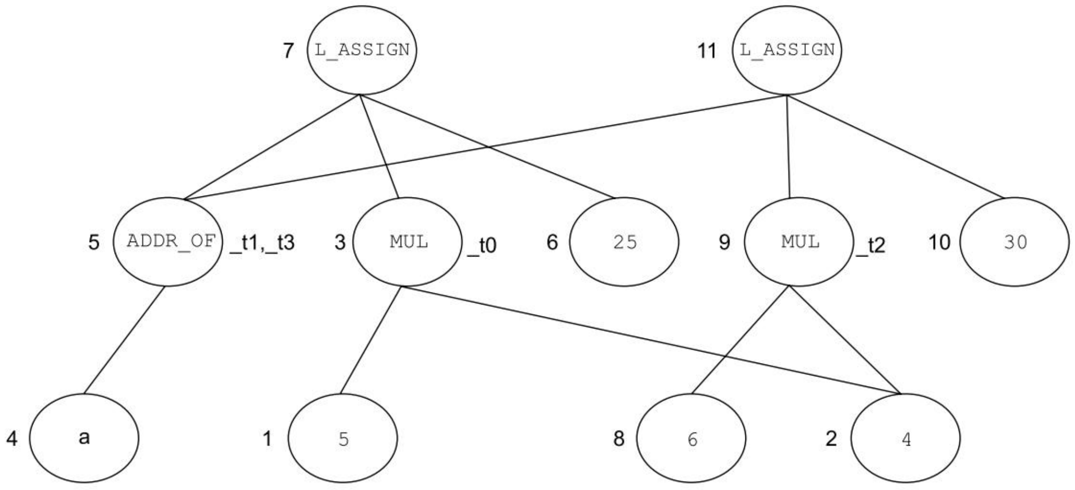


Fig. 7.25 *DAG*

In order to implement constant folding the DAG is traversed in the order of creation of the nodes. During the node traversal, if we find an interior node having class 1 operator (ADD, MUL, DIV, SUB, etc.) with both children as constants, we perform constant folding. For example, in the DAG we see in Fig. 7.25, the node 3 is an interior node (MUL) having two children, which are constants, namely 5 and 4. Node 3 is replaced by a leaf node whose value is 20 (result of 5 * 4). If the children do not have any other parents apart from the node in consideration and they do not have any attached user defined identifiers, they are removed from the DAG. The leaf node numbered 1 (label 5) does not have any other parents apart from node number 3, hence it is removed from the DAG. The other child, namely the node numbered 2 (label 4), is not removed, since it has another parent (node number 9). The DAG after making these changes is shown in Fig. 7.26.

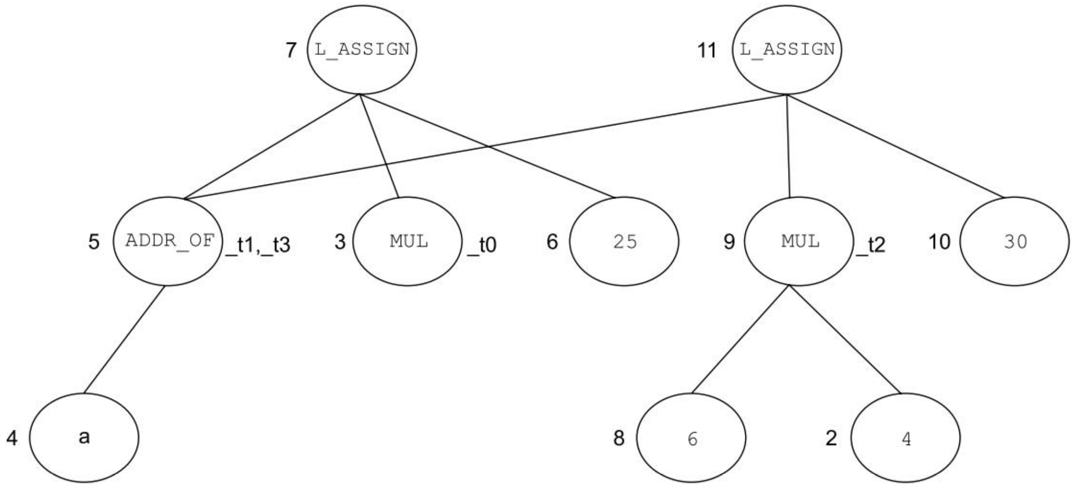


Fig. 7.26 *DAG after performing constant folding at node number 3*

In a similar way, the node numbered 9 is another candidate for performing constant folding, since both of its children are constants and the operator is MUL. The DAG after performing constant folding on node number 9 is shown in Fig. 7.27.

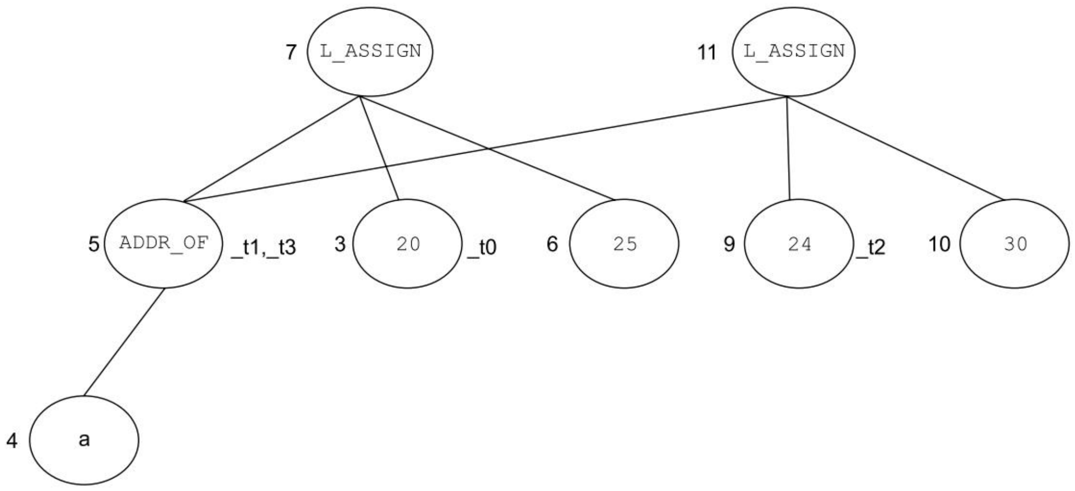


Fig. 7.27 *DAG after performing constant folding on node number 9*

The DAG in Fig. 7.27 is then traversed for generating the optimised quads as explained in Algorithm 7.3. The generated optimised quads for the DAG in Fig. 7.27 is shown below:

Table 7.33 *Regenerated quads*

Optimised code regenerated from DAG
(0) proc_begin main (1) _t1: = &arr1 (2) _t1[20] := 25 (3) _t1[24] := 30 (4) label .L0 (5) proc_end main

The above example illustrates how constant folding transformation can be performed on the unoptimised quads using the DAG.

Another optimising transformation—***dead store elimination*** can also be implemented using the DAG built for a basic block. Dead store elimination is implemented by traversing the DAG and removing root nodes that do not have any user defined variables attached to it. A root node is a node in DAG that does not have any parents.

Consider the code snippet and its corresponding intermediate code shown in Table 7.34 for understanding the elimination of dead store using the DAG. The DAG for the block 0 built using Algorithm 7.2 is shown in Fig. 7.28.

Table 7.34 *Input source and intermediate code*

Input source	TAC

`int a,b,c,d;`
`int x,y,z;`
 	
`void func ()`
`{`
`    x  = a + b; /* Dead Store */`
`    y  = b + c;`
`    x  = d + a;`
`}`
	`(1) proc_begin func`
	`(2) _t0: = a + b`
	`(3) x := _t0`
	`(4) _t1 := b + c`
	`(5) y := _t1`
	`(6) _t2 := d + a`
	`(7) x := _t2`
	`(8) label .L0`
	`(9) proc_end func`

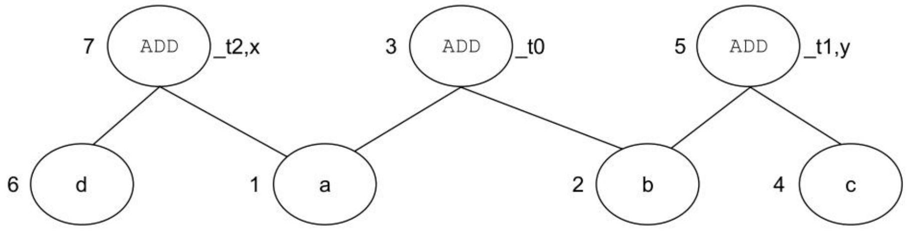


Fig. 7.28 *DAG*

In the DAG shown in Fig. 7.28 observe the node numbered 3. It has no parent. The list of identifiers attached to it contains of only one identifier _*t*0—a compiler-generated temporary that is not user defined. This node meets both the criteria of (1) not having any parent and (2) not having any user-defined variables attached to it. This node indicates a dead store and can be eliminated. The new DAG after the elimination of node 3 is shown in Fig. 7.29.

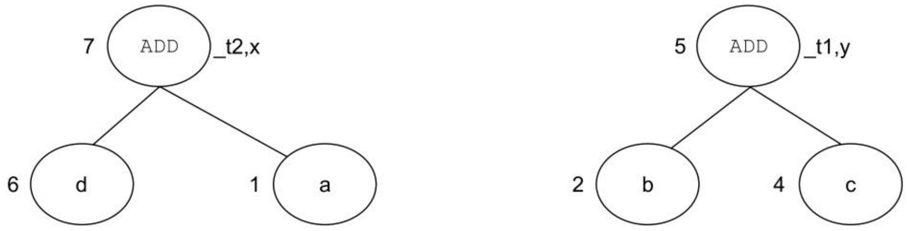


Fig. 7.29 *Modified DAG after elimination of node 3*

The quads for the function regenerated from the DAG in Fig. 7.29 are shown below.

Optimised code regenerated from DAG

`(0) proc_begin func`
`(1) y := b + c`
`(2) x := d + a`
`(3) label .L0`
`(4) proc_end func`

In the cases of programs containing multiple dead stores, repeated application of the above mentioned criteria in the DAG and removal of DAG nodes, eliminates all of the dead stores in the basic block.

To summarise, the process of making the DAG, revising it, and the subsequent regeneration of the optimised quads from the DAG helps in making the following optimising transformations within a basic block (a) common sub-expression elimination (b) copy propagation (c) removal of redundant assignments (d) constant folding and (e) dead store elimination.

7.2.9.8 Example 2—Local Optimisation using DAG This section demonstrates the toy C compiler (mycc) performing local optimisation of intermediate code by making the transformations like common sub-expression elimination, copy propagation, etc. The toy C compiler 'mycc' performs local optimisation by (a) constructing the DAG from the un-optimised TAC (Algorithm 7.2) and (b) regenerating the optimised quads from the DAG (Algorithm 7.3) as described in the preceding section.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC and (b) the locally optimised TAC. The dialog below shows 'mycc' taking in some sample input C sources, printing out unoptimised and locally optimised intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$ g++ -DCHAP7_EX2  -DICGEN -g -Wall ic_gen.cc optimise.cc target_code gen.cc mycc.cc
semantic_analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Common Sub-Expression Elimination Transformation
$ cat -n test2a.c
     1  /*
     2      Common Sub-expression
     3  */
     4  int a,b,c,d,e,f,g;
     5
     6  void func()
     7  {
     8
     9      int i,x;
    10
    11      a  = (b  + c)*d ;
    12      e  =  f * a ;
    13      f  = (b + c)*e;
    14      g  = d / (b + c);
    15
    16  }

$ ./mycc -i -O local -v test2a.c
```

```
TAC Before optimisation
(0) proc_begin func
(1) _t0: = b + c
(2) _t1: = _t0 * d
(3) a: = _t1
(4) _t2: = f * a
(5) e: = _t2
(6) _t3 := b + c
(7) _t4 := _t3 * e
(8) f: = _t4
(9) _t5 := b + c
(10) _t6 := d / _t5
(11) g := _t6
(12) label .L0
(13) proc_end func

TAC After Local Optimization
(0) proc_begin func
(1) _t0 := b + c
(2) a := _t0 * d
(3) e := f * a
(4) f := _t0 * e
(5) g := d / _t0
(6) label .L0
(7) proc_end func

# Copy Propagation Transform
$ cat -n test2b.c
    1  /*
    2      Copy Propagation
    3  */
    4
    5  int a,b,c,d,e,f,g;
    6
    7  void func()
    8  {
    9
   10      int i,x;
   11
   12      b  = a;
   13
   14      d  = (b + c)*f;
   15      e  = (a + c)*g ;
   16  }

$ ./mycc -i -O local -v test2b.c
TAC Before optimisation
(0) proc_begin func
(1) b := a
(2) _t0 := b + c
(3) _t1 := _t0 * f
(4) d := _t1
(5) _t2 := a + c
(6) _t3 := _t2 * g
(7) e := _t3
```

```
(8) label .L0
(9) proc_end func

TAC After Local Optimisation
(0) proc_begin func
(1) b := a
(2) _t0 := a + c
(3) d := _t0 * f
(4) e := _t0 * g
(5) label .L0
(6) proc_end func

# Constant Folding Transformation
$ cat -n test2c.c
    1  /*
    2      Constant Folding
    3  */
    4
    5  int a,b,c,d,e,f,g;
    6
    7  int func1()
    8  {
    9      int i;
   10
   11      b  = 5;
   12      a  = 6 ;
   13
   14      d  = f / (b * a) ;
   15      e  = g /f ;
   16
   17  }
   18

$ ./mycc -i -O local -v test2c.c
TAC Before optimisation
(0) proc_begin func1
(1) b := 5
(2) a := 6
(3) _t0 := b * a
(4) _t1 := f / _t0
(5) d := _t1
(6) _t2 := g / f
(7) e := _t2
(8) label .L0
(9) proc_end func1

TAC After Local Optimisation
(0) proc_begin func1
(1) b := 5
(2) a := 6
(3) d := f / 30
(4) e := g / f
(5) label .L0
(6) proc_end func1

# Dead Assignment Elimination Transformation
```

```
$ cat -n test2d.c
    1  /*
    2      Dead Store
    3  */
    4
    5  int a,b,c,d;
    6  int x,y,z;
    7
    8  void func ()
    9  {
   10
   11      x = a + b; /* Dead Store */
   12      y = b + c;
   13      x = d + a;
   14  }

$ ./mycc -i -O local -v test2d.c
TAC Before optimisation
(0) proc_begin func
(1) _t0 := a + b
(2) x := _t0
(3) _t1 := b + c
(4) y := _t1
(5) _t2 := d + a
(6) x := _t2
(7) label .L0
(8) proc_end func

TAC After Local Optimization
(0) proc_begin func
(1) y := b + c
(2) x := d + a
(3) label .L0
(4) proc_end func

# All of the transformations at Work
$ cat -n test2e.c
    1  /*
    2      All the Transformations at Work
    3      (1) Common Expression Elimination
    4      (2) Constant Propagation
    5      (3) Constant Folding
    6      (4) Dead Assignment Elimination
    7  */
    8  int a[45];
    9
   10  int func()
   11  {
   12      a[5]=25;
   13      a[6]=30;
   14  }
   15

$ ./mycc -i -O local -v test2e.c
TAC Before optimisation
(0) proc_begin func
```

```
(1) _t0 := 5 * 4
(2) _t1 := &a
(3) _t1[_t0] := 25
(4) _t2 := 6 * 4
(5) _t3 := &a
(6) _t3[_t2] := 30
(7) label .L0
(8) proc_end func

TAC After Local Optimization
(0) proc_begin func
(1) _t1 := &a
(2) _t1[20] := 25
(3) _t1[24] := 30
(4) label .L0
(5) proc_end func
```

7.2.9.9 ***Properties of DAG*** In this section, we discuss about some of the important properties of a DAG. We try and understand these properties by looking at the DAG construction process for a sample intermediate code.

Consider the input source and the TAC generated by the intermediate code generator in Table 7.35. There are several points that can be observed from Table 7.35.

(a) In the TAC, there is one basic block spanning from the quad (0) through (10). This basic block is essentially the core of the input source translated to the three address code.

(b) The values of '*y*1' and '*y*2' are used in the block (in quad 1) but are not computed in this block. The identifiers '*y*1' and '*y*2' are considered as ***input*** to the block.

(c) The value of '*x*1' computed in the quad (10) can be potentially used in a succeeding block, if any. The value of '*x*1' computed in the quad (2) cannot be used in the succeeding block, since it is overwritten in quad (10). The values of '*x*1', '*x*2', '*x*4' and '*x*3' computed at the quads 10, 9, 8 and 6 are considered as ***output*** from the block.

Table 7.35 *Input source and the intermediate code*

Input source	**TAC**
`int x1, x2, x3, x4, y1, y2;`	`(0) proc_begin func`
	`(1) _t0 := y1 + y2`
`void func()`	`(2) x1 := _t0`
`{`	`(3) _t1 := x1 + y2`
`    x1 = y1 + y2 ;`	`(4) x2 := _t1`
`    x2 = x1 + y2 ;`	`(5) _t2 := x1 + x2`
`    x3 = x1 + x2 ;`	`(6) x3 := _t2`
`    x4 = x1 + y2 ;`	`(7) _t3 := x1 + y2`
`    x2 = 0;`	`(8) x4 := _t3`
`    x1 = x3 ;`	`(9) x2 := 0`
`}`	`(10) x1 := x3`
	`(11) label .L0`
	`(12) proc_end func`

Figure 7.30 depicts the construction of DAG for the TAC shown above. There are several properties of the DAG that we can understand by looking at the DAG construction process and the nal DAG arrived at in step (D) of the gure.

(1) _t0 := y1 + y2
(2) x1 := _t0

(3) _t1 := x1 + y2
(4) x2 := _t1

(5) _t2 := x1 + x2
(6) x3 := _t2
(7) _t3 := x1 + y2
(8) x4 := _t3

(9) x2 := 0
(10) x1 := x3

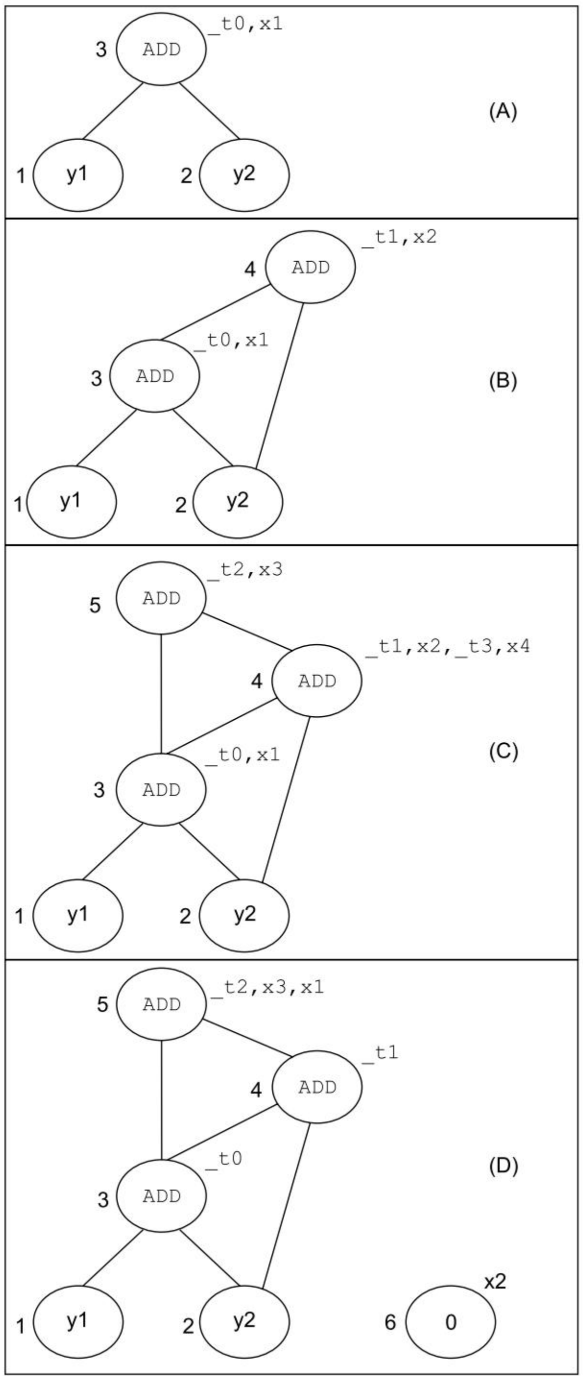


Fig. 7.30 *DAG construction process*

1. The nal DAG in (D) contains two leaves labeled '*y*1' and '*y*2'. These variables '*y*1' and '*y*2' are the input variables to the block that manifest as leaves in the DAG.
2. The variable '*x*3' was attached to node # 5 in (C) and it continued to be attached to it till the nal DAG in (D). The variable '*x*3' can be potentially used in the succeeding block if any, and it gets its nal value from node # 5. In a similar way, the variable '*x*4' was attached to node # 5 in (C) and it continued to be attached to it till the Final DAG in (D). The variable '*x*4' can be potentially used in the succeeding block if any, and it gets its nal value from node # 5. The variable '*x*1' is attached to node # 5 and variable '*x*2' is attached to node # 6 in step (D), which is also the final step in the DAG. The variables '*x*2' and '*x*3' can be potentially used in the succeeding block if any, and they derive their final value from node 5 and 6 respectively. The variables '*x*1', '*x*2', '*x*3' and '*x*4' are all examples of output variables identified by the fact that they are attached to a node at some stage in the DAG construction process and continue to stay attached till the final DAG.
3. The first evaluation of the sub-expression '*x*1 + *y*2' happened in step (B), when the node # 5 was created. The expression '*x*1 + *y*2' is identified as common during the DAG construction process, when in step (C) we have the variables '_*t*3' and '*x*4' being attached on node # 5. The common sub-expressions get identified in the process of constructing the DAG.
4. In a DAG, all the variables that hold the same values are all attached to the same node '*n*'. For example, _*t*2, *x*3 and *x*1 all hold the same value as given by node # 5 as seen from the final DAG. The reconstruction of TAC from the DAG uses this property in eliminating unnecessary assignments of the form '*x* := *y*'. The optimised TAC generated from the DAG contains no assignments to any of the temporary variables, unless they are necessary.

7.2.9.10 Arrays, Pointers and Procedure Calls in DAG The algorithms presented in the previous section for DAG construction and IC regeneration have some limitations with respect to handling of arrays, pointers and procedure calls in the input source. In this section, we identify those limitations and suggest improvements to the algorithms to handle them.

Arrays

In some of the cases of input source containing array references, the DAG construction and IC regeneration algorithms in the form explained previously result in incorrect generation of optimised code. In this section, we show an example of input source using array references, where the generated optimised code is incorrect. Later we look at the modification necessary in the DAG construction algorithm to fix the issue of incorrect generation of optimised code.

Consider the input source given in Table 7.36. It contains a sequence of array references—a read from an array in the form of arr[*i*], a write into array in the form of arr[*j*], followed by a read from array again in the form of arr[*i*] as seen in lines 6, 7 and 8 respectively. Table 7.36 also shows the intermediate code, before and after the local optimisation as given out by 'mycc' toy C compiler using the DAG construction and IC regeneration algorithms discussed in the previous sections.

Consider the optimised TAC instructions in Table 7.36. The quads (1) through (4) in the optimised TAC are correct. They represent the read of arr[*i*] into the identifier '*x*'. The quad (5) showing an assignment '*z* = *x*' is incorrect. We need to step back to the input source to understand why the quad (5) is incorrect. In-between the two reads of arr[*i*] at line 6 and line 8 in the input source, there is a possibility of a write at the location arr[*i*]. This happens when '*i*' and '*j*' have the same value. Thus, we cannot be assured that '*z*' and '*x*' would be equal all the time. Hence the quad (5) in the optimised TAC is incorrect. The correct code would need to read the value of arr[*i*] again and store it in '*z*'.

Table 7.36 *Input source and the intermediate code*

Input source	Un-optimised TAC	Optimised TAC

`int arr[50];`
`int x,y,z;`
 	
`int func(int i,int j)`
`{`
`   x  = arr[i];`
`   arr[j]=y;`
`   z  = arr[i];`
`}`
	`(0) proc_begin func`
	`(1) _t0 := i * 4`
	`(2) _t1 := &arr`
	`(3) _t2 := _t1[_t0]`
	`(4) x := _t2`
	`(5) _t3 := j * 4`
	`(6) _t4 := &arr`
	`(7) _t4[_t3] := y`
	`(8) _t5 := i * 4`
	`(9) _t6 := &arr`
	`(10) _t7 := _t6[_t5]`
	`(11) z := _t7`
	`(12) label .L0`
	`(13) proc_end func`
		`(0) proc_begin func`
		`(1) _t0 := i * 4`
		`(2) _t1 := &arr`
		`(3) x := _t1[_t0]`
		`(4) z := x`
		`(5) _t3 := j * 4`
		`(6) _t1[_t3]: = y`
		`(7) label .L0`
		`(8) proc_end func`

In order to appreciate why incorrect optimised TAC has been generated for the above example, we need to take a look at the DAG for the same. The DAG for the input source in Table 7.36 is shown in Fig. 7.31.

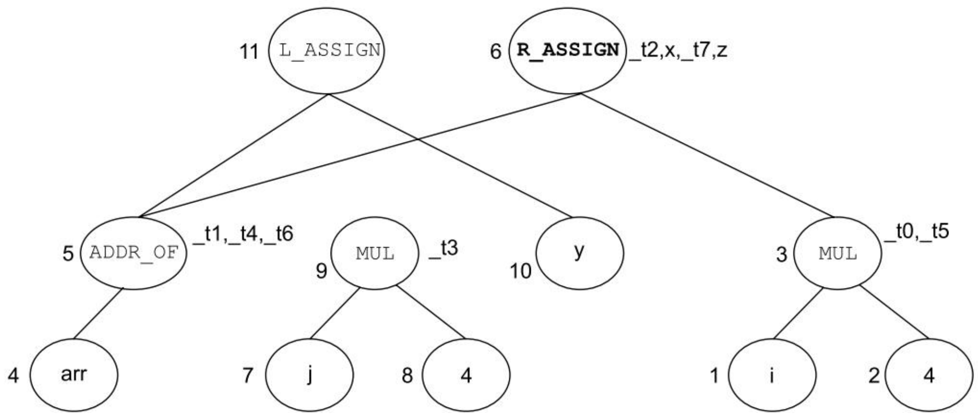


Fig. 7.31 *DAG using the original DAG construction algorithm*

Observe that the DAG construction process has identified arr[*i*] as a common expression and attached the identifiers *x* and *z* to it (see node #6). This is the genesis of incorrect code generation.

The algorithm that we used for constructing the DAG (Algorithm 7.2) needs to be modified to overcome this flaw and amend the generation of optimised code for array references. The revision in the algorithm is to process the assignments into arrays given by L_INDEX_ASSIGN of the form *a*[*b*] = *c* differently. In the modified version of the algorithm, at the time of processing of the quad with operator L_INDEX_ASSIGN, we mark all the nodes depending on the base address 'a' as ***'killed'***. The notion of killing a node is to make it ineligible to have any more identifiers attached to it. In other words, a killed node cannot be returned, when the algorithm looks for common sub-expression. This forces the DAG construction algorithm to create a new node on any further access of the array using '*a*' as the base address.

The DAG constructed by using the modified DAG construction algorithm for the same input source is shown in Fig. 7.32. It shows the node # 6 (shaded in gray) as killed, i.e. ineligible to have any more identifiers attached. The node # 6 is killed during the processing of the input TAC statement (7) _*t*4[_*t*3] : = *y* (L_INDEX_ASSIGN). The subsequent processing of the input statement (10) _*t*7 := _*t*6[_*t*5] creates a new node (node # 12), since node # 6 has been killed. Thus, we find that the _*t*7 and *z* are not attached to node #6 and falsely identified as a common sub-expression.

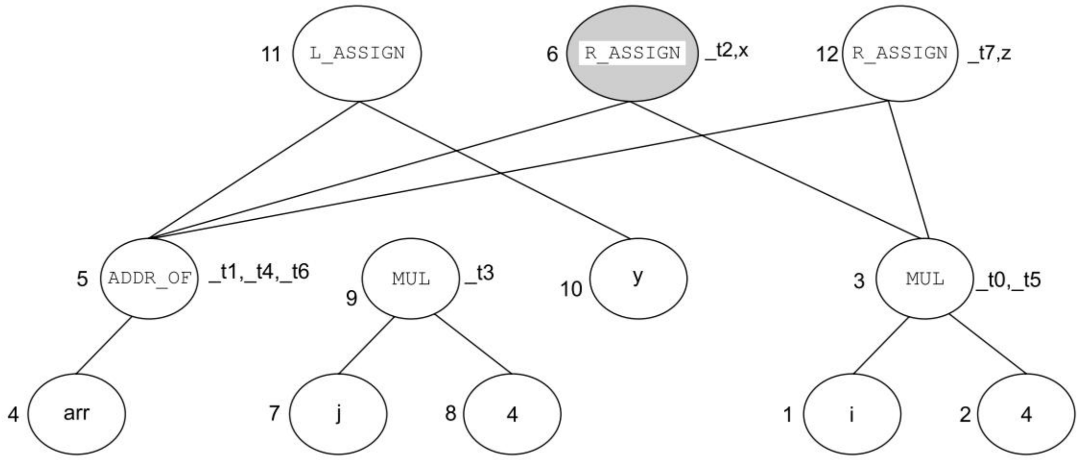
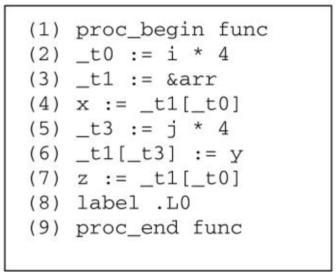


Fig. 7.32 *DAG using the modified DAG construction algorithm*

The optimised code generated from this DAG is given in Fig. 7.33. This set of quads is proper from the correctness standpoint.



```
(1) proc_begin func
(2) _t0  := i * 4
(3) _t1  := &arr
(4) x  := _t1[_t0]
(5) _t3  := j * 4
(6) _t1[_t3]  := y
(7) z  := _t1[_t0]
(8) label .L0
(9) proc_end func
```

Fig. 7.33 *Optimised quads*

Pointers

In some of the cases of input source containing usage of pointers, the DAG construction and IC regeneration algorithms in the form explained previously result in incorrect generation of optimised code. In this section, we show an example of input source using pointers, where the generated optimised code is incorrect. Later we look at the modification necessary in the DAG construction algorithm to fix the issue of incorrect generation of optimised code.

When there is a write into a variable using a pointer, the existing algorithm causes inaccuracies in the ***common sub-expression elimination*** leading to incorrect optimised code generation. A simple manifestation of write into a variable using a pointer and an incorrect identification of common sub-expression is shown in Table 7.37. The optimised TAC shown in Table 7.37 has incorrectly identified ‘*a* + *b*’ as a common sub-expression, since there is no explicit write into ‘*a*’ or ‘*b*’ in the code between the two computations of ‘*a* + *b*’. In reality, the variable ‘*a*’ is being written into indirectly by having a pointer to it.

Table 7.37 *Writing into variable via explicit pointer*

Input source	Un-optimised TAC	Optimised TAC

`int a,b,c,d;`
 	
`int fun ()`
`{`
 	
`      int *p;`
 	
`      c = a + b;`
`      p = &a ;`
`      *p = 100;`
 	
`      /* NOT cse */`
`      d = a + b;`
 	
`}`
	`(0) proc_begin fun`
	`(1) _t0 := a + b`
	`(2) c := _t0`
	`(3) _t1 := &a`
	`(4) p := _t1`
	`(5) p[0] := 100`
	`(6) _t2 := a + b`
	`(7) d := _t2`
	`(8) label .L0`
	`(9) proc_end fun`
		`(0) proc_begin fun`
		`(1) c := a + b`
		`(2) d := c`
		`(3) p := &a`
		`(4) p[0] := 100`
		`(5) label .L0`
		`(6) proc_end fun`

In the above case, the memory or the variable where the pointer was pointing to was clear from the code. In some other scenarios, it is difficult to pinpoint which variable is being written into or read from, especially in the cases where the control can flow in multiple paths. This type of ambiguity in the memory location that a pointer is pointing to (called ***ambiguous pointer***), also affects common sub-expression elimination in a similar fashion.

Consider the program shown in Table 7.38, which shows the input source, intermediate code, before and after the local optimisation as given out by the compiler using the DAG construction and IC regeneration algorithms that we had discussed in the previous sections.

Table 7.38 *Writing into memory location via ambigious pointer*

Input source	Un-optimised TAC	Optimised TAC

`int a,b,c,d;`
 	
`void fun(int x)`
`{`
`    int *p;`
 	
`    if(x> 10){`
`        p = &b;`
`    }else{`
`        p = &a;`
`    }`
 	
	`(0) proc_begin fun`
	`(1) if x > 10 goto .L0`
	`(2) goto .L1`
	`(3) label .L0`
	`(4) _t0: = &b`
	`(5) p := _t0`
	`(6) goto .L2`
	`(7) label .L1`
	`(8) _t1 := &a`
	`(9) p := _t1`
	`(10) label .L2`
	`(11) _t2 := a + b`
	`(12) c := _t2`
		`(0) proc_begin fun`
		`(1) if x > 10 goto .L0`
		`(2) goto .L1`
		`(3) label .L0`
		`(4) p := &b`
		`(5) goto .L2`
		`(6) label .L1`
		`(7) p := &a`
		`(8) label .L2`
		`(9) c := a + b`
		`(10) d := c`
		`(11) p[0] := 25`
		`(12) label .L3`

14 c = a + b;	(13) p[0] := 25	(13) proc_end fun
15	(14) _t3 := a + b	
16 /* a or b is overwritten */	(15) d := _t3	
17 *p = 25;	(16) label .L3	
18	(17) proc_end fun	
19 d = a + b; /* NOT cse */		
20		
21 }		
22		

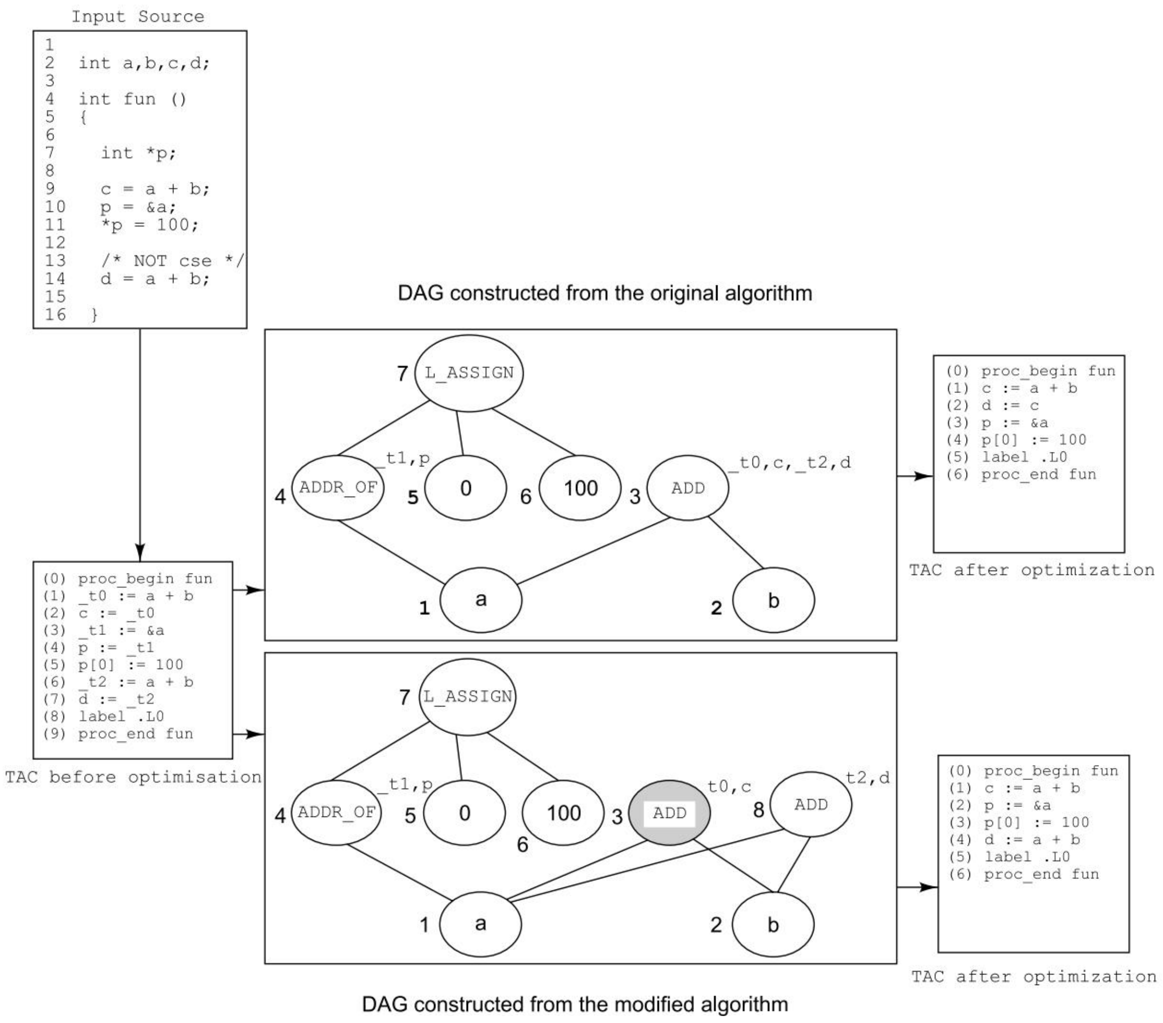
Observe that at line 17 of the input source, the pointer could be writing into '*a*' or '*b*' (ambiguous) depending on whether the flow of control was via line 9 or line 11. Since '*a*' or '*b*' could be overwritten, the evaluation of '*a* + *b*' in line 19 will be different from '*a* + *b*' in line 14. Hence, the expression '*a* + *b*' cannot be termed as a common sub-expression. The DAG construction algorithm discussed previously needs to be adapted to take care of write into variables pointed to using an *ambiguous* pointer such as '*p*' above.

In order to fix the above-mentioned issues with respect to incorrect identification of common sub-expression in the optimised code using pointers, the DAG construction algorithm is modified as follows. During a write into memory via an explicit pointer (L_INDEX_ASSIGN), the DAG construction algorithm identifies the variable '*v*', that is being written to by a pointer. The current node '*n*' associated with '*v*' is then found using get_current(). The parents of '*n*' are then 'killed' because they use the current value of '*v*', which is being overwritten via pointer. In cases where the pointer is ambiguous, we kill *all* the nodes that have an identifier attached. In both the cases, the killing of nodes forces an expression to be re-evaluated after the pointer is written into, resulting in correct optimal code.

Let's consider the examples we had seen above in Table 7.37 and Table 7.38, in which the generated optimised code, was incorrect. We shall revisit the same examples, this time with the above-mentioned modifications to the DAG construction algorithm and see how it helps us generate correct optimal code.

Consider the source code seen earlier in Table 7.37, for illustrating the incorrect generation of optimised code owing to a write using a pointer. Figure 7.34 shows the DAG constructed using the original and the modified DAG construction algorithm (as presented above) for the same input source. The corresponding optimised code generated from both of these DAGS is also shown. We can see from Fig. 7.34 that the node # 3 has been killed (shaded) in the case of DAG created from modified algorithm. This prevents the node # 3 from having more attached identifiers. The node # 3 is killed during the processing of TAC statement (5) *p*[0] := 100, (L_INDEX_ASSIGN) of the input TAC. During the processing of the statement (6) _*t*2 := *a* + *b*, the expression '*a* + *b*' is not considered as a common sub-expression because the node # 3 is already 'killed', i.e. made ineligible for attaching identifiers. This forces the creation of a new node # 8 for storing the expression '*a* + *b*'.

The case of handling a write using an ambiguous pointer is relatively simpler. A write using an ambiguous pointer (L_INDEX_ASSIGN) causes all the nodes having an attached identifier to be 'killed'. This prevents incorrect identification and elimination of common sub-expressions.



Input Source

```
int a,b,c,d;

int fun ()
{

  int *p;

  c = a + b;
  p = &a;
  *p = 100;

  /* NOT cse */
  d = a + b;

}
```

TAC before optimisation

```
(0) proc_begin fun
(1) _t0 := a + b
(2) c := _t0
(3) _t1 := &a
(4) p := _t1
(5) p[0] := 100
(6) _t2 := a + b
(7) d := _t2
(8) label .L0
(9) proc_end fun
```

TAC after optimization (DAG constructed from the original algorithm)

```
(0) proc_begin fun
(1) c := a + b
(2) d := c
(3) p := &a
(4) p[0] := 100
(5) label .L0
(6) proc_end fun
```

TAC after optimization (DAG constructed from the modified algorithm)

```
(0) proc_begin fun
(1) c := a + b
(2) p := &a
(3) p[0] := 100
(4) d := a + b
(5) label .L0
(6) proc_end fun
```

Fig. 7.34 *Write via pointer*

Procedure Calls

In some of the cases of input source containing calls to procedures, the DAG construction and IC regeneration algorithms in the form explained previously result in incorrect generation of optimised code. In this section, we look at some examples of input source using procedure calls, where the generated optimised code is incorrect. Later we look at the modification necessary in the DAG construction algorithm to fix the issue of incorrect generation of optimised code.

Consider the input source shown in Table 7.39. A pointer '*p*', pointing to the variable '*x*' is being passed into a function 'func1'. It is possible that the pointer '*p*' is de-referenced to either read or write into the variable '*x*' within the function 'func1'. The expression '*x* + *y*' cannot be a common sub-expression across the function invocation, since the value of '*x*' can be changed in 'func1'. The optimised code in Table 7.39, does not take this into account and judges '*x* + *y*' as a common sub-expression. The DAG construction algorithm needs to be modified to distinguish the variables that can be modified by means of call to a procedure and treat common sub-expressions accordingly. In languages like C, where the parameter passing is call-by-value, the only way a variable '*v*' in the caller can be modified by a callee is by passing a pointer to '*v*' as parameter to the callee. In C ++ , which supports both call-by-value and call-by-reference, care needs to be taken to ascertain which variable can be modified by the callee.

Table 7.39 *Effect of procedure call on CSE*

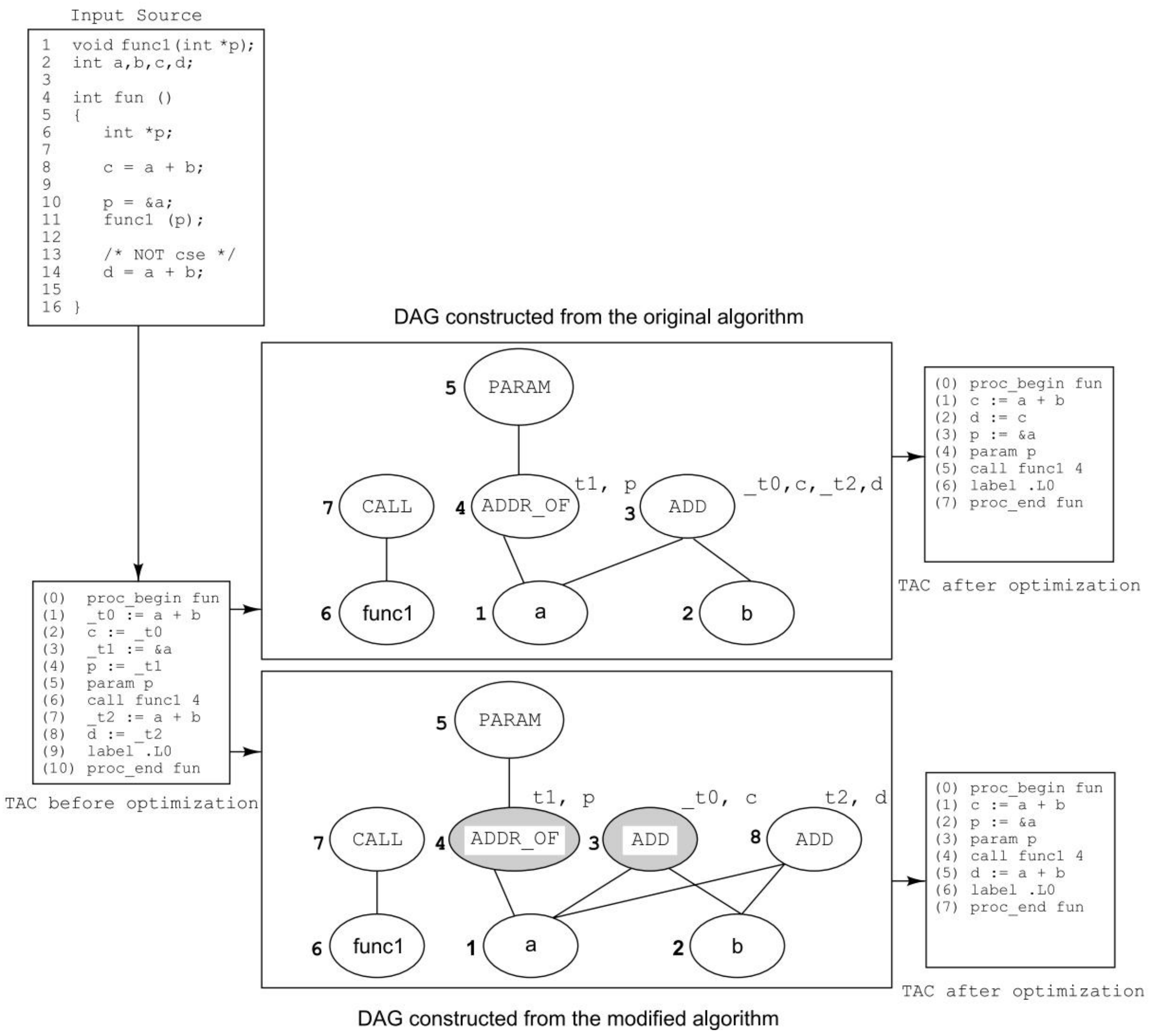
Input Source	Un-optimised TAC	Optimised TAC
`int func1(int *p);`	`(0) proc_begin func`	`(0) proc_begin func`
	`(1) _t0 := x + y`	`(1) a := x + y`
`int a,b;`	`(2) a := _t0`	`(2) b := a`
	`(3) _t1 := &x`	`(3) p := &x`
`void func(int x,int y)`	`(4) p := _t1`	`(4) param p`
`{`	`(5) param p`	`(5) call func1 4`
`int *p;`	`(6) call func1 4`	`(6) goto .L0`
	`(7) _t2 := x + y`	`(7) label .L0`
`a=x + y;`	`(8) b := _t2`	`(8) proc_end func`
	`(9) goto .L0`	
`p = &x;`	`(10) label .L0`	
	`(11) proc_end func`	
`/* can read/write`		
`into p */`		
`func1(p);`		
`/* NOT cse */`		
`b = x + y;`		
`return;`		
`}`		

In order to fix the above-mentioned issues in the optimised code related to the procedure calls, the DAG construction algorithm is modified in the following ways.

The processing of a call to a procedure (CALL statement) is modified to kill all the DAG nodes having an identifier attached to it. This prevents identification of common sub-expressions across a CALL statement. This also eliminates the possibility of identifying legitimate common expressions that span across a CALL statement. As we can see, the modification is similar to the modifications proposed for handling the erroneous generation of optimal code in the case of pointers.

We will now revisit the example shown in Table 7.39, with the improved algorithm and verify the optimised code for correctness. Consider the input source shown earlier in Table 7.39, for which the generated optimised code was incorrect. Figure 7.35 shows the DAG constructed using the original and the modified algorithm for the same input source. The corresponding optimised codes generated from both of these DAGS are also shown. We can see from Fig. 7.35 that the nodes #3 and # 4 have been killed (shaded) in the case of DAG created from modified algorithm. This prevents the node # 3 from having more attached identifiers. The node # 3 and # 4 (nodes with attached identifiers) were killed during the processing of TAC statement (6) call func1 in the input TAC. During the processing of the statement (7), the expression '*a* + *b*' is not considered as a common sub-expression because the node # 3 is already, 'killed', i.e. made ineligible for attaching identifiers. This forces the creation of a new node for storing the expression '*a* + *b*'. This is an illustration of the modification at work.

This concludes the discussion concerning the changes required in the DAG construction algorithm in the case of procedure calls.



Input Source

```
void func1(int *p);
int a,b,c,d;

int fun ()
{
   int *p;

   c = a + b;

   p = &a;
   func1 (p);

   /* NOT cse */
   d = a + b;

}
```

TAC before optimization

```
(0)  proc_begin fun
(1)  _t0 := a + b
(2)  c := _t0
(3)  _t1 := &a
(4)  p := _t1
(5)  param p
(6)  call func1 4
(7)  _t2 := a + b
(8)  d := _t2
(9)  label .L0
(10) proc_end fun
```

TAC after optimization

```
(0) proc_begin fun
(1) c := a + b
(2) d := c
(3) p := &a
(4) param p
(5) call func1 4
(6) label .L0
(7) proc_end fun
```

TAC after optimization

```
(0) proc_begin fun
(1) c := a + b
(2) p := &a
(3) param p
(4) call func1 4
(5) d := a + b
(6) label .L0
(7) proc_end fun
```

Fig. 7.35 *Optimisation with a CALL statement*

7.2.9.11 Example 3—Local Optimisation with DAG using the Improved Algorithm This section demonstrates the toy C compiler (mycc) performing local optimisation of intermediate code by using the improved DAG construction and TAC regeneration algorithms outlined in Section 7.2.9.10. The generated optimised intermediate code using the improved algorithms do not suffer from any of the deficiencies presented previously and is complete in terms of correctness.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC and (b) the locally optimised TAC. The dialog below shows 'mycc' taking in some sample input C sources containing arrays, pointers and procedure calls. It prints out the unoptimised and locally optimised intermediate code in TAC format as the output.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g + +  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g + +  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$  g++   -DICGEN  -g  -Wall  ic_gen.cc  optimise.cc  target_code_gen.cc  mycc.cc  semantic_
analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Sample Input C file containing array references
$ cat -n test3a.c
    1  int arr[50];
    2  int x,y,z;
    3
    4  int func(int i,int j)
    5  {
    6      x  = arr[i];
    7      arr[j]=y;
    8      z  = arr[i];
    9  }
   10

# Intermediate code before and after optimization
$ ./mycc.exe -i -O local -v test3a.c
TAC Before optimisation
(0) proc_begin func
(1) _t0 := i * 4
(2) _t1 := &arr
(3) _t2 := _t1[_t0]
(4) x := _t2
(5) _t3 := j * 4
(6) _t4 := &arr
(7) _t4[_t3] := y
(8) _t5 := i * 4
(9) _t6 := &arr
(10) _t7 := _t6[_t5]
(11) z := _t7
(12) label .L0
(13) proc_end func

TAC After Local Optimization
(0) proc_begin func
(1) _t0 := i * 4
(2) _t1 := &arr
(3) x := _t1[_t0]
(4) _t3 := j * 4
(5) _t1[_t3] := y
(6) _t5 := i * 4
(7) _t6 := &arr
```

```
(8) z: = _t6[_t5]
(9) label .L0
(10) proc_end func

# Sample Input C file with a write via pointer.Affects CSE
$ cat -n test3d.c
    1  int a,b,c,d;
    2
    3  int fun ()
    4  {
    5
    6      int *p;
    7
    8      c  = a + b;
    9      p  = &a ;
   10      *p  = 100;
   11
   12      /* NOT cse */
   13      d  = a + b;
   14
   15  }

# Intermediate code before and after optimization
$ ./mycc.exe -i  -O local -v test3d.c
TAC Before optimization
(0) proc_begin fun
(1) _t0 := a + b
(2) c := _t0
(3) _t1 := &a
(4) p := _t1
(5) p[0] := 100
(6) _t2 := a + b
(7) d := _t2
(8) label .L0
(9) proc_end fun

TAC After Local Optimization
(0) proc_begin fun
(1) c := a + b
(2) p := &a
(3) p[0] := 100
(4) d := a + b
(5) label .L0
(6) proc_end fun

# Sample Input C file using pointer to write into variable
# Ambiguous pointer. Affects CSE
$ cat -n test3f.c
    1
    2  int a,b,c,d;
    3
    4  void fun(int x)
    5  {
    6      int *p;
    7
    8      if(x> 10){
    9          p  = &b;
    10      }else{
    11          p  = &a;
    12      }
    13
    14      c  = a + b;
    15
    16      /* a or b is overwritten */
    17      *p  = 25;
    18
    19      d  = a + b; /* NOT cse */
    20
    21  }
    22

# Intermediate code before and after optimization
$ ./mycc.exe -i  -O local -v test3f.c
TAC Before optimization
(0) proc_begin fun
(1) if x > 10 goto .L0
(2) goto .L1
(3) label .L0
(4) _t0: = &b
(5) p := _t0
(6) goto .L2
(7) label .L1
(8) _t1 := &a
(9) p := _t1
(10) label .L2
(11) _t2 := a + b
(12) c := _t2
(13) p[0] := 25
(14) _t3 := a + b
(15) d := _t3
(16) label .L3
(17) proc_end fun

TAC After Local Optimization
(0) proc_begin fun
(1) if x > 10 goto .L0
(2) goto .L1
(3) label .L0
(4) p := &b
(5) goto .L2
(6) label .L1
(7) p := &a
(8) label .L2
(9) c := a + b
(10) p[0] := 25
(11) d := a + b
(12) label .L3
(13) proc_end fun

# Sample Input C file with a procedure call passing a pointer.
# The called procedure can write using the pointer. This Affects CSE.
$ cat -n test3h.c
     1  void func1(int *p);
     2
```

```
int a,b;

void func(int x,int y)
{
    int *p;

    a=x + y;

    p  = &x;

    /* can read/write
    into p */
    func1(p);

    /* NOT cse */
    b = x + y;

    return;
}

# Intermediate code before and after optimization
$ ./mycc.exe -i  -O local -v test3h.c
TAC Before optimization
(0) proc_begin func
(1) _t0 := x + y
(2) a := _t0
(3) _t1 := &x
(4) p : = _t1
(5) param p
(6) call func1 4
(7) _t2 : = x + y
(8) b : = _t2
(9) goto .L0
(10) label .L0
(11) proc_end func

TAC After Local Optimization
(0) proc_begin func
(1) a : = x + y
(2) p : = &x
(3) param p
(4) call func1 4
(5) b : = x + y
(6) goto .L0
(7) label .L0
(8) proc_end func
```

7.2.10 Global Optimisation

7.2.10.1 Introduction The global optimisation of the intermediate code is more complex than the local one, since it needs to take care of variety of issues like say branching in the flow of control, usage of variables across basic blocks, loop information, and so on, while constructing optimised code. We start the study of global optimisation for the intermediate code with discussion on some of the basic concepts in global optimisation.

Flow Graph

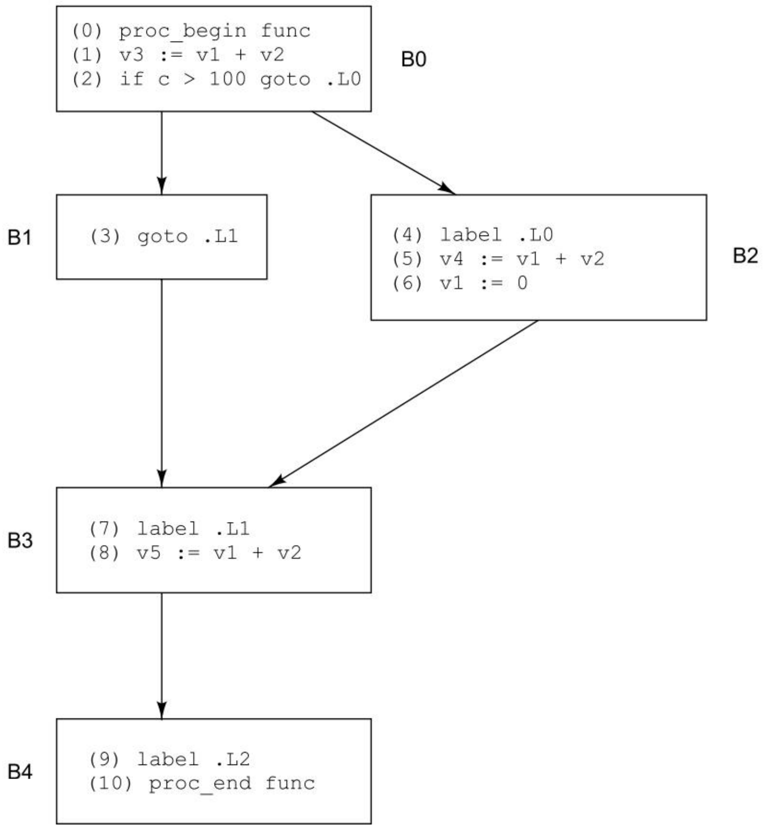
In local optimisation, the focus was essentially on the quads within a basic block, where the flow of control was in a straight line. In global optimisation, we are also concerned about how the flow of control goes from one block to another. The decisions taken for optimisation need to take into account all possible ways

```
int v1,v2,v3,v4,v5;

int func (int c)
{
   v3 = v1 + v2;

   if (c > 100){
       v4 = v1 + v2;
       v1 = 0;
   }

   v5 = v1 + v2;
}
```

Input source

```
(0) proc_begin func
(1) v3 := v1 + v2
(2) if c > 100 goto .L0
(3) goto .L1
(4) label .L0
(5) v4 := v1 + v2
(6) v1 := 0
(7) label .L1
(8) v5 := v1 + v2
(9) label .L2
(10) proc_end func
```

IC after local optimisation



```
B0:
(0) proc_begin func
(1) v3 := v1 + v2
(2) if c > 100 goto .L0

B1:
(3) goto .L1

B2:
(4) label .L0
(5) v4 := v1 + v2
(6) v1 := 0

B3:
(7) label .L1
(8) v5 := v1 + v2

B4:
(9) label .L2
(10) proc_end func
```

Flow graph

Fig. 7.36 *Input source, IC after local optimisation and the flow graph*

the control can flow. The information about the flow of the entire procedure, i.e. the possible blocks to get executed after a particular block B is represented in a graph called as the ***flow graph***. Each node in a flow graph is a basic block. In a flow graph, there is a directed edge from the block B1 and B2, only if (a) There is a conditional jump to the block B2 from the last quad of B1 or (b) the flow of control goes to B2 after B1 sequentially.

Figure 7.36 shows an input source, IC, after local optimisation and the corresponding flow graph. Remember that the input to global optimisation is the locally optimised intermediate code. The starting node in a flow graph is called as ***initial*** node. This is the basic block whose leader is the first TAC statement in the procedure. The node B0 in Fig. 7.36 is the initial node. The flow of control can go to block B1 or B2 from the block B0. The blocks B1 and B2 are called the ***successors*** of the block B0. Block B0 is called as a ***predecessor*** of block B1. B0 is called as the predecessor for block B2 also.

Point and Path

There are a couple of abstractions that are commonly used during the discussion on global optimisation, namely a ***point*** and ***path***.

A ***point*** is a place of reference that can be found at (a) before the first quad in a basic block (b) after the last quad in a basic block and (c) In between two quads within a basic block. Figure 7.37 shows the different points of the basic block 0 of the previous example. We can see that there are 4 points in basic block 0, given by *p0_b0, p1_b0, p2_b0* and *p3_b0*. We have suffixed the block number *b*0 to the point name to clearly identify that the point belongs to block 0. There are 3 quads and 4 points in the basic block 0. In general, there are (*i* + 1) points for a basic block containing '*i*' quads.

```
          ■   p0_b0
         (0)  proc_begin func
          ■   p1_b0
B0       (1)  v3  := v1 + v2
          ■   p2_b0
         (2)  if c > 100 goto .L0
          ■   p3_b0
```

Fig. 7.37 *Points*

Figure 7.38 shows a part of the flow graph containing the basic blocks B0, B1, B2 and B3 for the same example. It shows all the points in each of those basic blocks. A ***path*** is a sequence of points in which the control can flow. For example, there is a path between the points '*p0_b0*' and '*p6_b2*' given by the sequence of points, *p0_b0*, *p1_b0*, *p2_b0*, *p3_b0*, *p4_b2*, *p5_b2* and *p6_b2*. There is no path between *p3_b1* and *p6_b2*, since there is no sequence of points that can take the control from *p3_b1* to *p6_b2*. Observe that there are two possible paths from *p0_b0* to *p7_b3*. The path1 consists of the sequence of points, *p0_b0*, *p1_b0*, *p2_b0*, *p3_b0*, *p3_b1*, *p4_b1*, and *p7_b3*. The path2 consists of the sequence of points, *p0_b0*, *p1_b0*, *p2_b0*, *p3_b0*, *p4_b2*, *p5_b2*, *p6_b2*, *p7_b2*, and *p7_b3*. It is not common to illustrate each of the points in the flow graph, the reader needs to absorb the idea of the points and path from a given flow graph.

Definition and Usage of Variables

The concepts of definition and usage of variables are used extensively during the study of global optimisation. Consider the input source and the TAC (locally optimised) shown in Table 7.40 to understand those twin concepts. The quad (1) ***defines*** the variable '*v*3'. In other words, the quad (1) assigns a value to the variable '*v*3'. The quad(1) ***uses*** the variables '*v*1' and '*v*2' to define '*v*3'. Similarly, quad (5) defines '*v*4' and uses '*v*1' and '*v*2'.

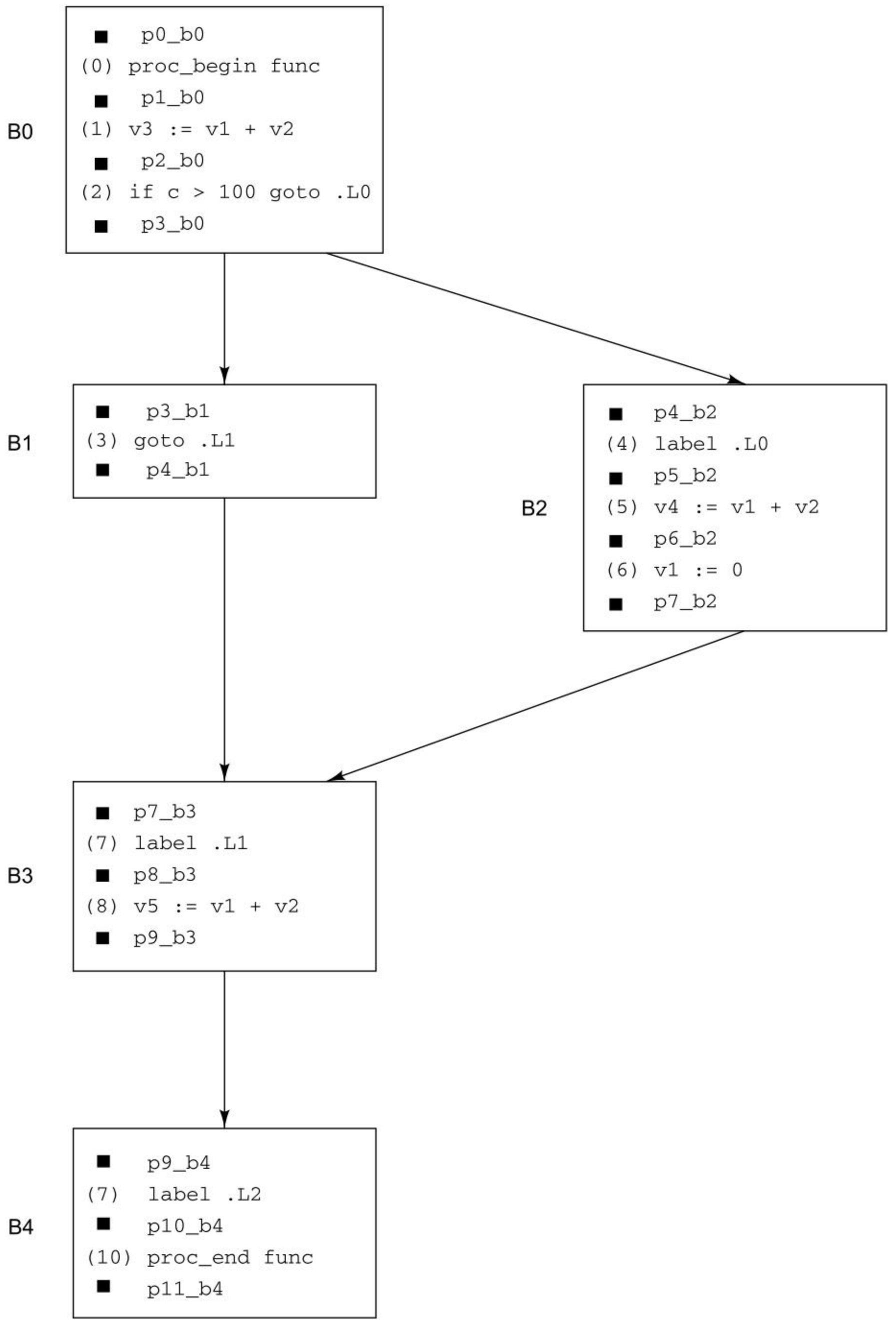


Fig. 7.38 *Points and paths*

Table 7.40 *Define and use*

Input source	Locally optimised TAC
`int v1,v2,v3,v4,v5;`	`(0) proc_begin func`
	`(1) v3: = v1 + v2`
`int func(int c)`	`(2) if c > 100 goto .L0`
`{`	`(3) goto .L1`
`    v3  = v1 + v2 ;`	`(4) label .L0`
	`(5) v4 := v1 + v2`
`    if(c > 100){`	`(6) v1 := 0`
`        v4  = v1 + v2;`	`(7) label .L1`
`        v1  = 0;`	`(8) v5 := v1 + v2`
`    }`	`(9) label .L2`
	`(10) proc_end func`
`    v5  = v1 + v2 ;`	
`}`	

In the above example, the assignment to '*v*3' in quad (1) is explicit. There would be cases, where a quad defines a variable implicitly through a pointer. As an example, consider the input source and the corresponding three-address code in Table 7.41. In this case, the variable '*x*' is defined in the quad (3) implicitly by using the pointer '*p*'. We can say that quad (3) defines the variable '*x*' assertively, since we know that the pointer '*p*' is unambiguously pointing to the variable '*x*' at the time when control reaches quad (3).

Table 7.41 *Define via unambiguous pointer*

Input source	Locally optimised TAC
`int x,y,z;`	`(0) proc_begin func`
	`(1) p := &x`
`int func()`	`(2) _t1 := y + z`
`{`	`(3) p[0] := _t1`
`    int *p;`	`(4) label .L0`
	`(5) proc_end func`
`    p  = &x;`	
`    /* x is assigned */`	
`    *p  = y + z;`	
`}`	

Table 7.42 presents another case, where the pointer '*p*' is ambiguous, i.e. we cannot say that the quad (10) defines '*a*' or '*b*' assertively, since it depends on how the control has passed before it came to the quad (10). In such cases where there is a write into a variable via an ambiguous pointer, it is considered to have defined all the variables that are present in the scope.

Table 7.42 *Defining via ambiguous pointer*

Input source	Locally optimised TAC
`int a,b,c,d;`	`(0) proc_begin fun`
	`(1) if x > 10 goto .L0`
`void fun(int x)`	`(2) goto .L1`

{	(3) label .L0
int *p;	(4) p : = &b
	(5) goto .L2
if(x> 10){	(6) label .L1
p = &b;	(7) p := &a
}else{	(8) label .L2
p = &a;	(9) _t2 := c + d
}	(10) p[0] := _t2
	(11) label .L3
/* a or b is assigned */	(12) proc_end fun
*p = c + d;	
}	

It is also possible for a procedure call to define a variable '*v*' by passing it as a parameter using the pass-by-reference scheme.

Data Flow Analysis

The global optimisation is performed after the local optimisation. The local optimisation would typically have eliminated the common sub-expressions, removed dead code, performed copy propagation and other optimisations mentioned in Section 7.3.9 at a block level. The global optimiser takes the quads that come out of the local optimiser and works on optimising them at a global level across the blocks.

The global optimiser elicits a variety of information from the input quads in order to make informed decisions during optimisation. It uses specialised algorithms to analyse the input quads and extract useful details so that it can perform the optimisation of the quads.

Let's take an example to understand the kind of information that is extracted from the input quads to perform optimisation at a global level. Consider the input source and the intermediate code after local optimisation shown in Table 7.43. The expression '*v*1 + *v*2' has been computed in the quad (1) in the basic block 0. When the control comes to quad (5), we can re-use the computed value of the expression '*v*1 + *v*2' from the quad (1). Observe that the values of '*v*1' and '*v*2' have not changed from the time '*v*1 + *v*2' has been computed at (1) till the time the control reaches (5). In other words, in order to eliminate the re-computation of the common sub-expression '*v*1 + *v*2' in a global fashion at (5), it is required to figure out the information that the expression '*v*1 + *v*2' has been computed earlier in a different block and is readily 'available' for use. The local optimisation would have taken care of eliminating a common sub-expression, if it were within the same block. The global optimisation needs to analyse the quads in all of the basic blocks to absorb the information about which expressions are 'available' and at what point. Observe that we cannot eliminate the re-computation of the expression '*v*1 + *v*2' at the quad(8), because it is not 'available' at that point.

Table 7.43 *Input source and the TAC after local optimisation*

Input source	TAC after local optimisation
int v1,v2,v3,v4,v5;	(0) proc_begin func
	(1) v3 := v1 + v2
int func(int c)	(2) if c > 100 goto .L0
{	
v3 = v1 + v2 ;	(3) goto .L1

```
    if(c > 100){

    /* 'v1 + v2' is available
            here */
        v4 = v1+v2;

         /* CSE  Killed */
      v1 = 0;
    }

    /* 'v1+v2' is NOT available
         here */
    v5 = v1 + v2 ;
}
```

```
(4) label .L0
(5) v4 := v1 + v2
(6) v1 := 0

(7) label .L1
(8) v5 := v1 + v2

(9) label .L2
(10) proc_end func
```

The information about the 'available expressions' that we saw in the above example is used by the global optimiser in order to eliminate common sub-expressions at a global level. There are many other such pieces of information like say 'reaching definitions', 'liveness', and so on that the global optimiser requires for performing different types of optimisations. We study a few of them in the forthcoming sections.

The analysis of the input quads for obtaining information such as the 'available expressions' in the above example is known as ***data flow analysis***. By using data flow analysis, we collect specific information about a ***data flow property***. In the above example, we have computed the data flow property commonly called as 'available expressions'. The algorithm used for collecting a data flow property say 'available expressions' might differ vastly from the one used for collecting another data flow property say 'reaching definitions'.

In the next few sections, we study about some of the data flow properties used commonly during global optimisation. Each of those sections is structured to discuss about (a) introduction to the data flow property (b) the algorithm that is used to collect the particular data flow property (c) how the data flow property is used to make a particular global optimisation, and (d) example demonstrating the global optimisation using the data flow property.

7.2.10.2 *Available Expressions*

Introduction There was a brief introduction to the data flow property called 'available expressions' in the Section 7.2.10.1. We study about that in detail in this section. Available expressions (AE) is a data flow property that is computed by the global optimiser using data flow analysis for eliminating the re-evaluation of common sub-expressions globally across blocks.

Let's start off with a formal definition of 'available expression' using the notion of points and path that we studied in Section 7.2.10.1. An expression '$a + b$' is available at a point 'p', if both of the following conditions are satisfied.

(a) Every path from initial node in the flow graph to 'p' evaluates '$a + b$'.

(b) After the last such evaluation of '$a + b$' and before reaching 'p' in every path, there are no subsequent assignments to 'a' or 'b'.

Let's use the example in Table 7.43 and check whether the expression '$v1 + v2$' is available at the points just before quad (5) and quad (8) using the above definition. The quads (5) and (8) re-compute the expression '$v1 + v2$', which can be avoided in case the expression is available at those points.

Figure 7.39 shows the flow graph of the Intermediate code after local optimisation annotated with the individual points for the discussion.

Let's check the availability of expression '*v*1 + *v*2' at the point '*p*5_*b*2' using the above definition of 'available expression'. There is only one path—path1 leading to *p*5_*b*2 from the initial node, which is given by the sequence of points *p*0_*b*0, *p*1_*b*0, *p*2_*b*0, *p*3_*b*0, *p*4_*b*2 and *p*5_*b*2. The last evaluation of '*v*1 + *v*2' happens at quad (1). There is no assignment to '*v*1' or '*v*2' after the last evaluation, along the sequence of points in path1 mentioned above. Hence, the expression '*v*1 + *v*2' is available at the point *p*5_*b*2.

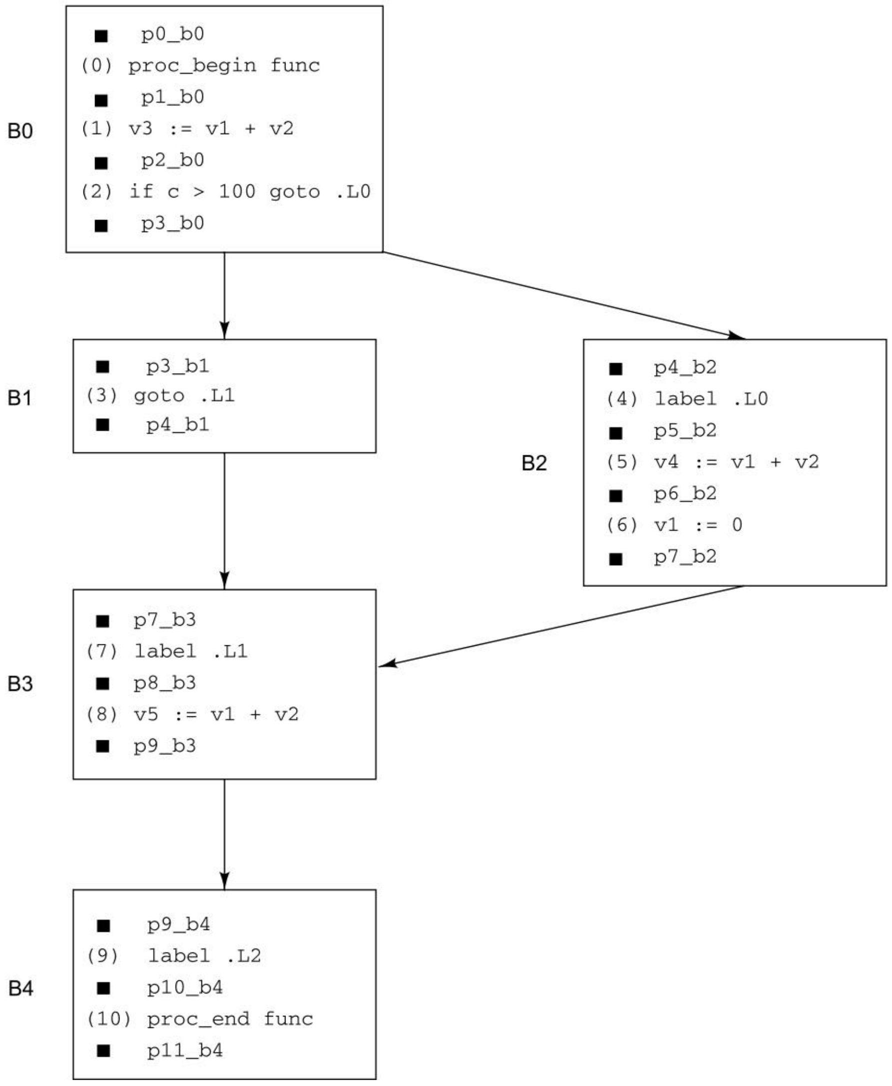


Fig. 7.39 *Flow graph with points*

Let's check the availability of expression '*v*1 + *v*2' at the point '*p*8_*b*3' using the above definition of 'available expression'. There are 2 paths leading to *p*8_*b*3 from the initial node. The path1 is given by the sequence of points, *p*0_*b*0, *p*1_*b*0, *p*2_*b*0, *p*3_*b*0, *p*3_*b*1, *p*4_*b*1, *p*7_*b*3 and *p*8_*b*3. The last evaluation of ‘*v*1 + *v*2’ happens at quad (1) in this path. There is no assignment to ‘*v*1’ or ‘*v*2’ after the last evaluation along the sequence of points in path1 mentioned above. The path2 is given by *p*1_*b*0, *p*2_*b*0, *p*3_*b*0, *p*3_*b*1, *p*4_*b*2, *p*5_*b*2, *p*6_*b*2, *p*7_*b*2, *p*7_*b*3 and *p*8_*b*3. The last evaluation of ‘*v*1 + *v*2’ happens at quad (5) in this path. There is an assignment to ‘*v*1’ at quad (6) after the last evaluation, in this path. This violates the condition (b) mentioned above for the path2. Hence, the expression ‘*v*1 + *v*2’ is not available at the point *p*8_*b*3.

From the above discussion, the expression ‘*v*1 + *v*2’, is available at the point *p*5_*b*2, so it can be used to eliminate the re-evaluation of common sub-expression at quad (5). Table 7.44 shows the TAC after local optimisation and the TAC after eliminating the global common sub-expression—*v*1 + *v*2. We can observe that the re-computation of the expression ‘*v*1 + *v*2’ has been avoided in the quad (5). The quads of relevance for the elimination of the global common sub-expression ‘*v*1 + *v*2’ are shaded in gray in Table 7.44. This is a testimony of how the information on AE is used for elimination of global common sub-expressions.

Table 7.44 *Global common sub-expression elimination using AE*

Input source	TAC after local optimisation	TAC after elimination of global common sub-expression
`1    int v1,v2,v3,v4,v5;`	`(0) proc_begin func`	`(0) proc_begin func`
`2`	`(1) v3 := v1 + v2`	`(1) _t3 := v1 + v2`
`3    int func(int c)`	`(2) if c > 100 goto .L0`	`(1a) v3 := _t3`
`4    {`		`(2) if c > 100 goto .L0`
`5        v3  = v1 + v2 ;`	`(3) goto .L1`	`(3) goto .L1`
`6`		`(4) label .L0`
`7        if(c > 100){`	`(4) label .L0`	`(5) v4 := _t3`
`8`	`(5) v4 := v1 + v2`	`(6) v1 := 0`
`9          /* ‘v1+v2’ is available`	`(6) v1 := 0`	`(7) label .L1`
`10           here */`		`(8) v5 := v1 + v2`
`11         v4  = v1 + v2;`	`(7) label .L1`	`(9) proc_end func`
`12`	`(8) v5 : = v1 + v2`	
`13         /* CSE  Killed */`		
`14         v1  = 0;`	`(9) label .L2`	
`15       }`	`(10) proc_end func`	
`16`		
`17       /* ‘v1+v2’ is NOT available`		
`18          here */`		
`19       v5  = v1 + v2 ;`		
`20    }`		

Table 7.45 illustrates more examples with information on the available expressions in order to enhance the understanding of the same. The description here uses the input source to make conclusions on the available expressions, just for the ease of understanding. In practice, the global optimiser elicits the available expression information from the quads post local optimisation using the algorithms described in the next sections.

Table 7.45 *Input sources and the available expressions*

Input source	Comments on the available expressions
`1   int v1,v2,v3,v4,v5;`	The expression 'v1 + v2' is available at line 10, since the only path that reaches line #10 (5->7->10) evaluates 'v1 + v2' at line #5 and there are no subsequent assignments to 'v1' or 'v2' between line #5 and line #10. The line #20 can be reached in 2 ways. One of the paths reaching line #20 is (5->7->10->13->16->20) evaluates 'v1 + v2' at line #16 and there are no subsequent assignments to 'v1' or 'v2' between line #16 and line #20 in this path. The other path to line #20 is 5->7->20, evaluates 'v1 + v2' at line #5 and there are no subsequent assignments to 'v1' or 'v2' between line #5 and #20 in this path. Hence the expression 'v1 + v2' is available at line #20.
`2`	
`3   int func(int c)`	
`4   {`	
`5       v3  = v1 + v2 ;`	
`6`	
`7       if( c > 100){`	
`8`	
`9           /* 'v1+v2' is available here as CSE */`	
`10          v4 = v1 + v2;`	
`11`	
`12           /* CSE  Killed */`	
`13          v1  = 0;`	
`14`	
`15          /* computing 'v1+v2' again */`	
`16          v6  = v1 + v2;`	
`17      }`	
`18`	
`19      /* 'v1+v2' is available here as CSE */`	
`20      v5  = v1 + v2 ;`	
`21  }`	
`1   int v1,v2,v3,v4,v5,v6;`	The line #12 can be reached in 2 ways. One of the paths reaching line #12 is (5->6->12) evaluates 'v1 + v2' at line #6 and there are no subsequent assignments to 'v1' or 'v2' between line #6 and line #12 in this path. The other path to line #12 is 5->8->12, evaluates 'v1 + v2' at line #8 and there are no subsequent assignments to 'v1' or 'v2' between line #8 and #12 in this path. The expression 'v1 + v2' is available at line #12, since both the conditions (a) and (b) mentioned above are satisfied.
`2`	
`3   int func(int c)`	
`4   {`	
`5       if(c > 100){`	
`6           v3 = v1 + v2;`	
`7       }else{`	
`8           v4 = v1 + v2 ;`	
`9       }`	
`10`	
`11      /* 'v1+v2' is available here as CSE */`	
`12      v5 = v1 + v2 ;`	
`13  }`	

Data Flow Analysis to Compute AE in Intermediate Code We now study about the algorithms that can be used for computing the AE properties of a given TAC and subsequently use it for globally eliminating common sub-expression.

We start off with some terminology.

A block ***generates*** an expression say '*a* + *b*', if it evaluates '*a* + *b*' and does not make any assignment to '*a*' or '*b*' subsequently in the block. We use the term e_GEN[B] to denote the expressions generated by a basic block B. For example, the term e_GEN[B0] represents the expressions generated by block 0.

A block ***kills*** an expression say '*a* + *b*', if it assigns a value to '*a*' or '*b*' and does not subsequently re-compute the value '*a* + *b*'. We use the term e_KILL[B] to denote the expressions killed by a basic block B. For example, the term e_KILL[B0] represents the expressions killed by block 0.

The generation and killing of expressions are used in tandem with 'L' a universal set of expressions appearing on the right side of one or more quads of the program spanning across all the basic blocks.

Consider the sample TAC shown in Table 7.46 to understand how the universal set of expressions L, is computed. The universal set of expressions L consists of any expression that appears on the right side of one or more statements. By inspecting the input quads in Table 7.46, we can see that the expression '*a* + *b*' is used in quad (1). The expression '*b* * *c*' is used in the quad (2) The expression '*c* + *d*' is used in (3) and (8). The expression '*a* * *b*' is used in (5) and (9) The expression '*b*/*c*' is used in quad (6). Thus, we have the universal set of expressions L = {*a* + *b*, *b* * *c*, *c* + *d*, *a* * *b*, *b*/*c*}.

Table 7.46 *Sample TAC*

```
 (0)  proc_begin test_func
 (1)  x   = a + b
 (2)  y   = b * c
 (3)  d   = c + d
 (4)  if u > 100 goto .L1

 (5)  x   = a * b
 (6)  y   = b / c
 (7)  b   = 45
 (8)  z   = c + d
 (9)  u   = a * b

(10)  label .L1
(11)  proc_end test_func
```

In order to calculate the set of generated expressions e_GEN[B], we need to consider the points in a block from the beginning to the end of block. For a quad '*x* := *y* + *z*', sandwiched between two points '*p*' and '*q*' as shown in Fig. 7.40, the set of generated expressions at the point '*q*' would be calculated from e_GEN, the set of generated expression at p, as follows:

(a) Add the expression '*y* + *z*' to the set e_GEN.

(b) Delete from e_GEN any expression that uses '*x*'.

The steps (a) and (b) would have be done in the same order for catering to situations where the operand '*x*' on the LHS is the same as one of the operands on the RHS—'*y*' or '*z*'. The quad #3 in Table 7.46 is an example where the operand on the LHS is '*d*' and one of the operands on the right is also '*d*'.

```
p
(n) x := y + z
q
```

Fig. 7.40 *A quad between two points*

Let's take the example of basic block shown in Table 7.47 and calculate the set of generated expressions at the end of the block using the steps (a) and (b) mentioned above. Table 7.47 illustrates the TAC along with the points that need to be considered for the calculation of generated expressions.

Table 7.47 *Points in the basic block 0*

```
p0_b0
(0) proc_begin test_func
p1_b0
(1) x  = a + b
p2_b0
(2) y  = b * c
p3_b0
(3) d  = c + d
p4_b0
(4) if u > 100 goto .L1
p5_b0
```

Table 7.48 shows the step-by-step computation of e_GEN for the basic block 0.

Table 7.48 *Computing the e_GEN for Block 0*

#	TAC under consideration	Explanation

1	p0_b0
	(0) proc_begin test_func
	p1_b0
		At the point p0_b0, the start of the initial block there are no generated expressions, so e_GEN[p0_b0] = { ø }.
		
		No expressions are used or killed in quad (0) , hence
		e_GEN[p1_b0] = e_GEN[p0_b0]
		e_GEN[p1_b0] = { ø }

2	p1_b0
	(1) x = a + b
	p2_b0
		e_GEN[p1_b0] = { ø } as computed above.
		
		step(a)
		e_GEN[p2_b0] = e_GEN[p1_b0] + { a+b }
		e_GEN[p2_b0] = { a+b }
		
		step(b)
		There are no expressions in e_GEN[p2_b0] containing 'x'to be deleted , so it remains as is.
		
		e_GEN[p2_b0] = { a+b }

3	p2_b0
	(2) y = b * c
	p3_b0
		e_GEN[p2_b0] = { a+b } as computed above.
		
		step(a)
		e_GEN[p3_b0] = e_GEN[p2_b0] + { b*c}
		e_GEN[p3_b0] = { a+b, b*c }
		
		step(b)
		There are no expressions in e_GEN[p3_b0] containing 'y'to be deleted , so it remains as is.
		
		e_GEN[p3_b0] = { a+b, b*c }

4	p3_b0
	(3) d = c + d
	p4_b0
		e_GEN[p3_b0] = { a+b, b*c } as computed above.
		
		step(a)
		e_GEN[p4_b0] = e_GEN[p3_b0] + { c+d }
		e_GEN[p4_b0] = { a+b, b*c , c+d }
		
		step(b)
		There is an expression in e_GEN[p4_b0] containing 'd' which is c+d. We now,delete c+d from e_GEN[p4_b0].
		
		e_GEN[p4_b0] = { a+b, b*c, c+d } – { c+d }
		e_GEN[p4_b0] = { a+b, b*c }

5	p4_b0
	(4) if u > 100 goto .L1
	p5_b0
		e_GEN[p4_b0] = { a+b, b*c } as computed above.
		
		step(a) and (b)
		No expressions are used or killed in quad (4) , hence
		e_GEN[p5_b0] = e_GEN[p4_b0]
		e_GEN[p5_b0] = { a+b, b*c }
		
		The value of e_GEN[B0] is the same as e_GEN[p5_b0], since it is the last point in the block.
		
		The value of e_GEN[B0] = { a+b,b*c } is the set of Generated expressions for the Block 0.

The generated set of expressions e_GEN[B0] for the block 0 as calculated above is $= \{a + b, b * c\}$.

The set of killed expressions e_KILL, is all the expressions ‘$y + z$’, which satisfy the following criteria:

(a) Either y or z are defined in the block.

(b) The expression ‘$y + z$’ is not generated by the block.

Let’s now calculate the set of killed expressions for the block 0 in the above example. The universal set of expressions L, used in the TAC is $= \{a + b, b * c, c + d, a * b, b/c\}$. This was calculated previously. In the block 0, we define ‘x’ in quad (1), ‘y’ in quad (2) and ‘d’ in quad (3). In L, the expressions that involve x, y, or d are $\{c + d\}$. The expression ‘$c + d$’ is not generated by block 0. Hence, the set of killed expressions in block 0, e_KILL[B0] is $= \{c + d\}$.

Figure 7.41 shows the universal set of expressions L, e_GEN and e_KILL for the block 0 as calculated in this discussion.

```
L = {a+ b, a + d, b + c, c + d}

(0) proc_begin test_func
(1) x = a + b
(2) y = b * c                  e_GEN[B0] = {a + b, b * c}
(3) d = c + d                  e_KILL[B0] = {c + d}
(4) if u > 100 goto .L1
```

Fig. 7.41 *e_GEN and e_KILL for block 0*

The reader is encouraged to calculate the values of e_GEN and e_KILL for the other blocks.

The data flow properties like AE and the others that we study in the next few sections are computed using equations know as ***data flow equations***. By using the data flow equations, one can compute the values of data flow property like AE at the block boundaries. The value of data flow property is then extrapolated to the point of interest within the block by using its value at the block boundary.

The data flow equations for AE express the relationship between the ***e_IN[B]***, the set of expressions that are available at the beginning of block ‘B’ to ***e_OUT[B]*** the set of expressions that are available at the end of block ‘B’. The data flow equations for available expressions are as follows:

e_OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]) e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block e_IN[B0] = ø where B0 is the initial block	Equation 7.1

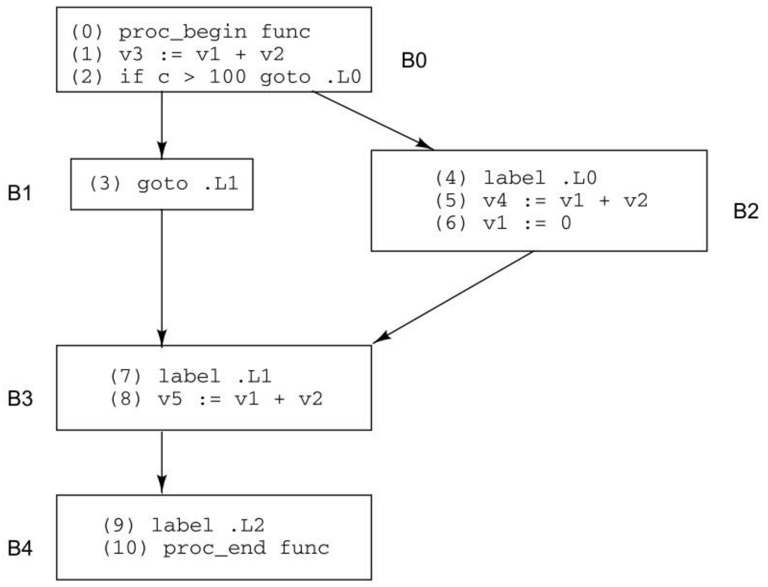
Let’s use the data flow equations for available expressions given by Equation 7.1 and calculate the available expressions at the start (e_IN[B]) and end of each block (e_out[B]) for the sample code shown in Fig. 7.36 reproduced below for convenience. There are 5 basic blocks (B0–B4) each of them is seen as a node in the flow graph.

```
int v1,v2,v3,v4,v5;

int func (int c)
{
   v3 = v1 + v2;

   if (c > 100){
      v4 = v1 + v2;
      v1 = 0;
   }

   v5 = v1 + v2;
}
```

Input source

```
(0) proc_begin func
(1) v3 := v1 + v2
(2) if c > 100 goto .L0
(3) goto .L1
(4) label .L0
(5) v4 := v1 + v2
(6) v1 := 0
(7) label .L1
(8) v5 := v1 + v2
(9) label .L2
(10) proc_end func
```

IC after local optimisation



Flow graph

The following table shows the calculation of the sets e_IN and e_OUT for all the blocks B0 through B4 using the data flow equations defined by Equation 7.2.

Table 7.49 *Computation of e_IN and e_OUT for the basic blocks*

Block No.	TAC	e_IN	e_OUT	Comments
B0	(0) proc_begin func (1) v3 := v1 + v2 (2) if c > 100 goto .L0	**{ø}**	**{v1 + v2}**	The universal set of expressions that are used in the entire procedure is given by L = {v1 + v2} e_IN[B0] = ø as defined by the Data Flow Equation in Equation 7.1 e_GEN[B0] = {v1 + v2}, since the block evaluates 'v1 + v2' and does not make any assignment to 'v1' or 'v2' subsequently in the block

				e_KILL[B0] = {ø} since there is no assignment to v1 or v2, which can kill the only expression 'v1 + v2' in L.
				
				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]), we have
				e_OUT[B0] = {v1 + v2} U ({ø} – {ø})
				e_OUT[B0] = {v1 + v2}

B1	(3) goto .L1	{v1 + v2}	{v1 + v2}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block e_IN[B1] = e_OUT[B0], since there is only one predeccesor i.e. B0 Hence, e_IN[B1] = {v1 + v2} e_GEN[B0] = {ø}, since the block does not evaluate any expression e_KILL[B0] = {ø} since there is no assignment to v1 or v2, which can kill the expression 'v1 + v2' From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]), we have e_OUT[B1] = {ø} U ({v1 + v2} – {ø}) e_OUT[B1] = {v1 + v2}
B2	(4) label .L0 (5) v4 := v1 + v2 (6) v1 := 0	{v1 + v2}	{ø}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block e_IN[B2] = e_OUT[B0], since there is only one predecessor for B2 i.e. B0 Hence, e_IN[B2] = {v1 + v2} e_GEN[B2] = {ø}, since the block evaluates expression 'v1 + v2', but subsequently has an assignment to v1. e_KILL[B2] = {v1 + v2} since there is an assignment to v1, which can kill the expression 'v1 + v2' From the data flow equation OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]), we have e_OUT[B2] = {ø} U ({v1 + v2} – {v1 + v2}) e_OUT[B2] = {ø}
B3	(7) label .L1 (8) v5 := v1 + v2	{ø}	{v1 + v2}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block

				e_IN[B3] = e_OUT[B1] ∩ e_OUT[B2], since there are two predecessors for B3 i.e. B1 and B2
				Hence, e_IN[B3] = {v1 + v2} ∩ {ø}
				e_IN[B3] = {ø}
				
				e_GEN[B3] = {v1 + v2} since the block evaluates expression 'v1 + v2', and it has no assignment to v1 or v2 later.
				
				e_KILL[B3] = {ø} since there is no assignment to v1 or v2, which can kill the expression 'v1 + v2'
				
				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] - e_KILL[B]), we have
				e_OUT[B3] = {v1 + v2} U ({ø} - {ø})
				e_OUT[B3] = {v1 + v2} U {ø}
				e_OUT[B3] = {v1 + v2}

B4	(9) label .L2
	(10) proc_end func
		{v1 + v2}	{v1 + v2}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block
				
				e_IN[B4] = e_OUT[B3], since there is only one predecessor for B4 i.e. B3
				Hence, e_IN[B4] = {v1 + v2}
				e_IN[B4] = {v1 + v2}
				
				e_GEN[B4] = {ø} since the block evaluates no expression.
				
				e_KILL[B4] = {ø} since there is no assignment to v1 or v2, which can kill the expression 'v1 + v2'
				
				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] - e_KILL[B]), we have
				e_OUT[B4] = {ø} U ({v1 + v2} - {ø})
				e_OUT[B4] = {ø} U {v1 + v2}
				e_OUT[B4] = {v1 + v2}

Figure 7.42 shows the flow graph annotated with the values of e_IN, e_GEN, e_KILL and e_OUT for each of the blocks—B0 through B4.

In this example, we have been able to compute the available expression information at the start of the block in the form of e_IN by using the Equation 7.1 for all the blocks in the program. We have been able to compute the values of e_IN and e_OUT for each of the blocks by proceeding block after block in the same order as the data flow, i.e. B0, followed B1, B2, B3, and B4. We had to strictly follow the order of flow of control while computing the e_IN because the e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block. We could not have computed e_IN[B], even if the e_OUT of one of the predecessors were not yet computed.

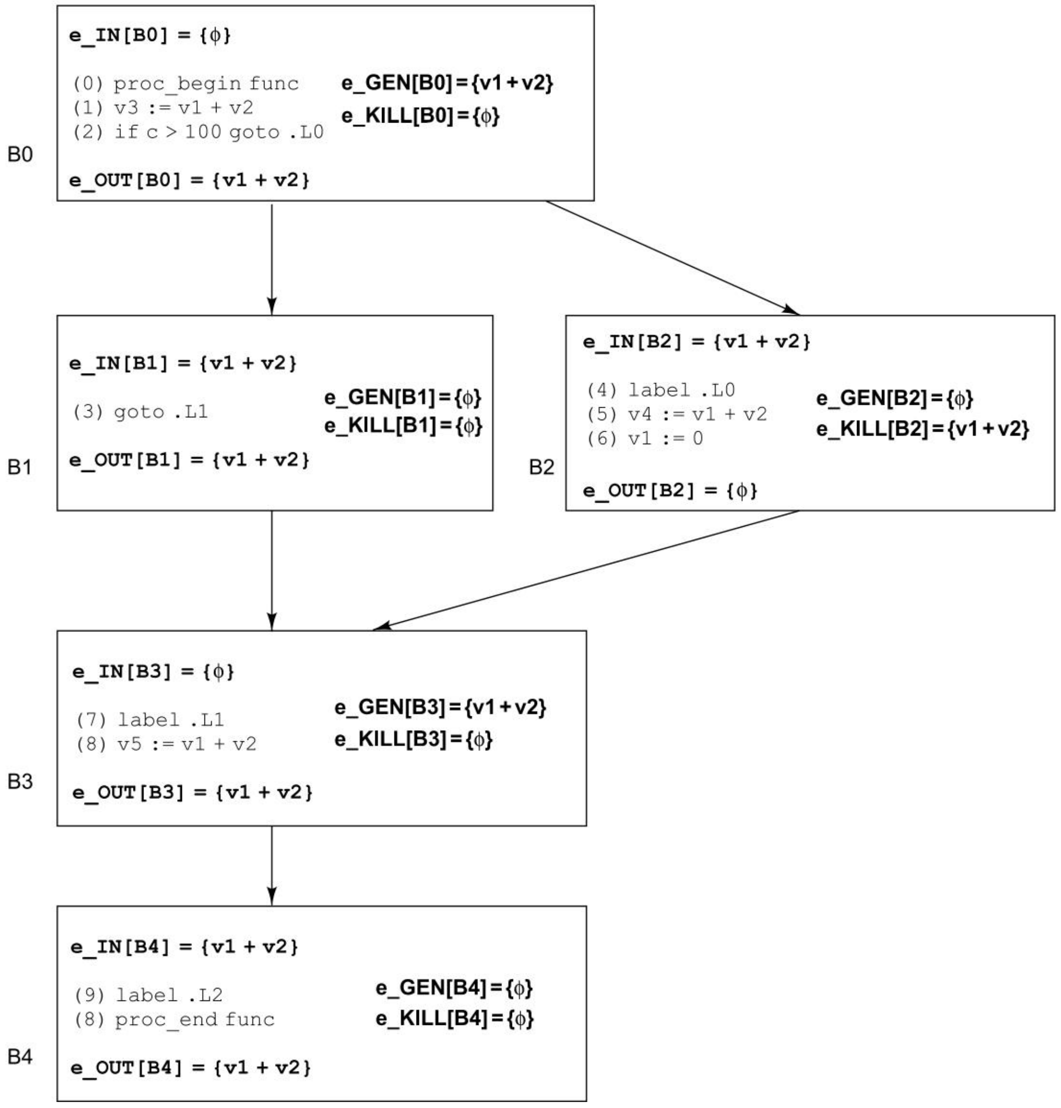


Fig. 7.42 *Flow graph annotated with available expression information*

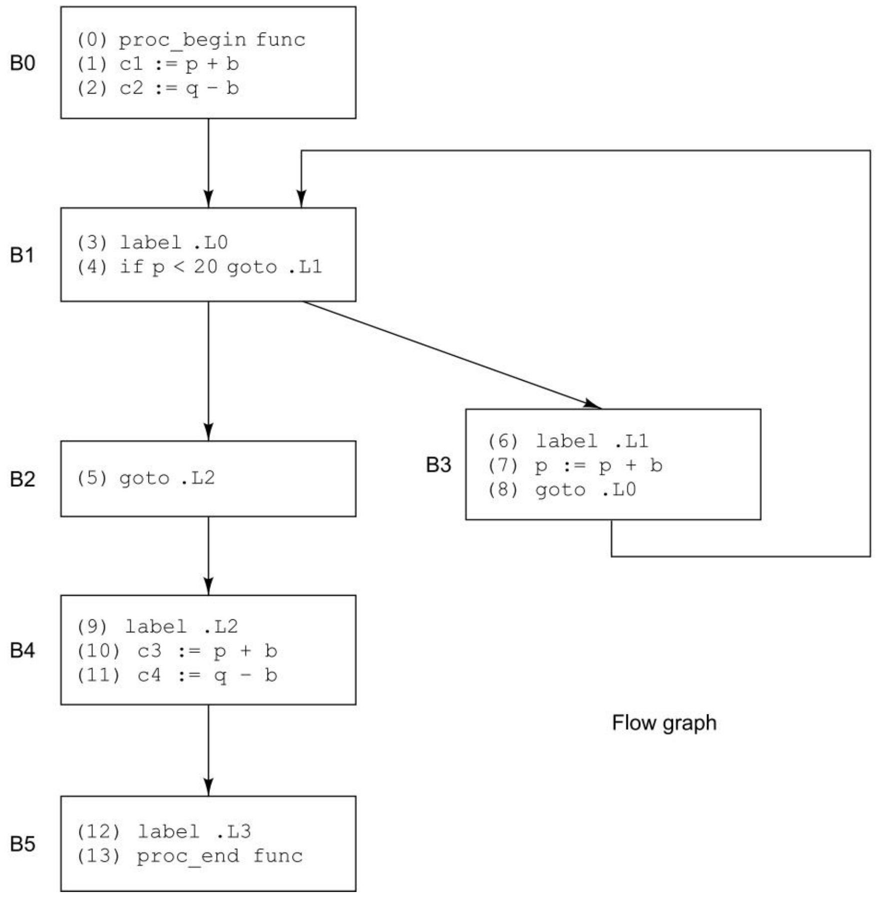
Let's now consider a scenario for computing the available expression (AE) in the form of e_IN and e_OUT in an input source involving loops. We shall use the data flow equations for available expressions given by Equation 7.1 and calculate the available expressions at the start (e_IN[B]) and end of each block e_OUT[B] for the sample code shown in Fig. 7.43. Figure 7.43 also shows the corresponding TAC after local optimisation and its flow graph. There are 6 basic blocks (B0–B5) each of them is seen as a node in the flow graph.

```

int c1, c2, c3 c4;

int p, q;

int func(int a, in b)
{

   /* CSE */
   c1 = p + b;
   c2 = q - b;

   while (p < 20){
       p = p + b;
   }

   c3 = p + b;
   c4 = q - b;

}
```

Input source

```
(0) proc_begin func
(1) c1 := p + b
(2) c2 := q - b

(3) label .L0
(4) if p < 20 goto .L1

(5) goto .L2

(6) label .L1
(7) p := p + b
(8) goto .L0

(9) label .L2
(10) c3 := p + b
(11) c4 := q - b

(12) label .L3
(13) proc_end func
```

IC after local optimisation



```
B0:
(0) proc_begin func
(1) c1 := p + b
(2) c2 := q - b

B1:
(3) label .L0
(4) if p < 20 goto .L1

B2:
(5) goto .L2

B3:
(6) label .L1
(7) p := p + b
(8) goto .L0

B4:
(9) label .L2
(10) c3 := p + b
(11) c4 := q - b

B5:
(12) label .L3
(13) proc_end func
```

Flow graph

Fig. 7.43 *Input source, IC after local optimisation and the flow graph*

The computation of e_IN and e_OUT for the basic blocks is described in Table 7.50.

Table 7.50 *Computation of e_IN and e_OUT for the basic blocks*

Block No.	TAC	e_IN	e_OUT	Comments

B0	`(0) proc_begin func`
	`(1) c1 := p + b`
	`(2) c2 := q - b`
		{ø}	{p + b, q - b}	The universal set of expressions that are used in the entire procedure is given by L = {p + b, q - b}
				
				e_IN[B0] = ø as defined by Equation 7.1
				
				e_GEN[B0] = {p + b, q - b}, since the block evaluates 'p + b' and 'q - b'. There are no subsequent assignments to 'p' or 'q' or 'b' subsequently in the block
				
				e_KILL[B0] = {ø} since There is no assignment to p or b, which can kill the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q - b'.
				
				From the data flow equation OUT[B] = e_GEN[B] U (e_IN[B] - e_KILL[B]), we have OUT[B0] = {p + b, q - b} U ({ø} - {ø})
				e_OUT[B0] = {p + b, q - b}

Let's now look at the calculation of e_IN and e_OUT of the block B1, which offers some challenges. The e_IN for the block is defined by Equation 7.1

$$e\_IN[B] = \cap\, e\_OUT[P] \text{ for all the predecessors P of the block}$$

The block B1 has two predecessors B0 and B3, hence the e_IN for B1 can be written as

$$e\_IN[B1] = e\_OUT[B0] \cap e\_OUT[B3].$$

The e_OUT for B0 has been calculated above. However, the e_OUT of B3 is not available at this point for the computation of e_IN[B1]. We cannot compute the value of e_OUT[B3] at this point either, because we would require the e_OUT[B1] to be known for the calculation of e_OUT[B3], since B1 is one of the predecessor for the block B3. This type of cyclic dependency between the nodes of the flow graph makes it difficult to calculate e_IN at some of the nodes like B1 in the cases of input source having loops.

This issue is overcome by having an ***iterative approach*** to solving the data flow equations for the available expressions given by Equation 7.1. In the iterative approach of solving data flow equations for available expressions, an initial value for e_OUT[B] for every block B is provided. This will help us compute the e_IN[B], where e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block, even in cases where e_OUT[P] is not yet computed, like the block 3 in the above example. The initial value of e_OUT[B] for every block B that is assigned before the start of the first iteration is (L – e_KILL[B]). Observe that this initial value, (L – e_KILL[B]) is the maximum value that e_OUT can take. In other words the e_OUT[B] is initialised with the maximum possible value. For the initial block, we initialise e_IN[B0] to ø and e_OUT[B0] to e_GEN[B0].

Now, using the value of e_IN[B], we compute e_OUT[B] for each one of the blocks B, with the help of the equation e_OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]). In this manner, we compute e_IN and e_OUT for all the blocks. Observe that the e_IN of some of the blocks like B1, might have been computed directly based on the initial value of e_OUT for one of the predecessor — B3, while the e_IN of other blocks like B2, B4, etc. are calculated based on the computed values of e_OUT of the predecessors. At the end of one round of computation for all the blocks (let's call it iteration 1), we have a set of values of e_IN/e_OUT for all the blocks. At this point, note that the e_OUT for every block contains the computed value and not the initial value that we assigned at the start. Table 7.51 shows e_IN/e_OUT calculation for the first iteration corresponding to all the nodes (basic blocks) of the flow graph in Fig. 7.43.

Next, we do the computation of e_IN and e_OUT for all the blocks again (say iteration 2) using the same equations defined in Equation 7.1. In this second iteration, the value of e_IN for the blocks like B1 might change, since e_IN[B1] = e_OUT[B0] ∩ e_OUT[B3], and e_OUT[B3] would now be the value computed in the first iteration as opposed to initial value. The changes in e_IN[B1] could have a ripple effect changing e_OUT[B1], e_IN[B3] and then e_OUT[B3]. Thus, the second iteration yields a set of values of e_IN/e_OUT for all the blocks.

We calculate the e_IN / e_OUT for all the blocks again in the third iteration, fourth iteration and so on. We stop the iterations, when we see that for every block, the value of e_OUT has not changed, when compared to its value in the previous iteration. In other words, we stop the iterations, when the e_OUT for all the blocks have reached a steady state. The Table 7.52 shows the values of e_IN/e_OUT as calculated in the first and second iteration corresponding to all the nodes (basic blocks) of the flow graph in Fig. 7.43. Observe that the values of e_OUT for none of the blocks have changed from the first to second iteration. This signals us to halt the iterations calculating e_IN/e_OUT for the blocks, since e_OUT of all the blocks have reached a steady state.

Table 7.51 shows the calculation of e_IN/e_OUT for each of the blocks in the first iteration using the initial values of e_OUT, wherever required.

Table 7.51 *Computation of e_IN and e_OUT in the first iteration*

#	TAC	e_IN	e_OUT	Comments
B0	(0) proc_begin func (1) c1 := p + b (2) c2 := q - b	{ø}	{p + b, q - b}	The universal set of expressions that are used in the entire procedure is given by L = {p + b, q - b} **e_IN[B0] = ø** as defined in Equation 7.1 e_GEN[B0] = {p + b, q - b}, since the block evaluates 'p + b' and 'q - b'. There are no subsequent assignments to 'p' or 'q' or 'b' subsequently in the block

				e_KILL[B0] = {ø} since
				There is no assignment to p or b, which can kill the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q − b'.
				
				From the data flow equation OUT[B] = e_GEN[B] U (e_IN[B] − e_KILL[B]), we have
				OUT[B0] = {p + b, q − b} U ({ø} − {ø})
				e_OUT[B0] = {p + b, q − b}

B1	(3) label .L0
	(4) if p < 20 goto .L1
		{q − b}	{q − b}	According to Equation 7.1 we have e_IN[B] =∩ e_OUT[P] for all the predecessors P of the block
				
				e_IN[B1] = e_OUT[B0] ∩ e_OUT [B3], since there are 2 predecessors i.e. B0 and B3
				
				We have already computed **e_OUT[B0] = {**p + b, q − b**} in the previous step.**
				
				e_OUT[B3] is initialised with (L − e_KILL[B3])
				L = {p + b, q − b}
				e_KILL[B3] = {p + b} since there is an assignment to 'p', but there is subsequent re-computation of p + b. There are no assignments to 'q' or 'b', which can kill the expression 'q − b'.
				
				e_OUT[B3] = (L − e_KILL[B3])
				e_OUT[B3] = ({p + b, q − b} − {p + b})
				e_OUT[B3] = {q − b}
				
				e_IN[B1] = e_OUT[B0] ∩ e_OUT[B3]
				e_IN[B1] = {p + b, q − b} ∩ {q − b}
				e_IN[B1] = {q − b}
				
				e_GEN[B1] = {ø}, since the block does not evaluate 'p + b' and 'q − b'.
				
				e_KILL[B1] = {ø} since
				There is no assignment to p or b, which can kill the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q − b'.
				
				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] − e_KILL[B]), we have
				
				e_OUT[B1] = {ø} U ({q − b} − {ø})
				e_OUT[B1] = {q − b}

B2	(5) goto .L2	{q − b}	{q − b}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block e_IN[B2] = e_OUT[B1], since there is only 1 predecessor i.e. B1 We have already computed **e_OUT[B1] = {**q − b**} in the previous step.** e_IN[B2] = e_OUT[B1] e_IN[B2] = {q − b} e_GEN[B2] = {ø}, since the block does not evaluate 'p + b' and 'q − b'. e_KILL[B2] = {ø} since There is no assignment to p or b, which can kill the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q − b'. From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] − e_KILL[B]), we have e_OUT[B2] = {ø} U ({q − b} − {ø}) e_OUT[B2] = {q − b}
B3	(6) label .L1 (7) p := p + b (8) goto .L0	{q − b}	{q − b}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block e_IN[B3] = e_OUT[B2], since there is only 1 predecessor i.e. B2 We have already computed **e_OUT[B2] = {**q − b**} in the previous step.** e_IN[B3] = e_OUT[B2] e_IN[B3] = {q − b}
				e_GEN[B3] = {ø}, since the block evaluates 'p+b', but assigns it to 'p'. e_KILL[B3] = {p + b} since There is assignment to p, which kills the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q−b'. From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] − e_KILL[B]), we have

				e_OUT[B3] = {ø} U ({q − b} − {p + b})
				e_OUT[B3] = {ø} U ({q − b})
				e_OUT[B3] = {q − b}

B4	(9) label .L2
	(10) c3 := p + b
	(11) c4 := q − b
		{q − b}	{p + b, q − b}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block.
				
				e_IN[B4] = e_OUT[B3], since there is only 1 predecessor, i.e. B3
				
				We have already computed **e_OUT[B3] = {**q − b**} in the previous step.**
				
				e_IN[B4] = e_OUT[B3]
				e_IN[B4] = {q − b}
				
				e_GEN[B4] = {p + b, q − b}, since the block evaluates 'p + b' and 'q − b'.
				
				e_KILL[B4] = {ø} since
				There is no assignment to p, which kills the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q−b'.
				
				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] − e_KILL[B]), we have
				
				e_OUT[B4] = {p + b, q − b} U ({q − b} − {ø})
				e_OUT[B4] = {p + b, q − b} U ({q − b})
				e_OUT[B4] = {p + b, q − b}

B5	(12) label .L3 (13) proc_end func	{p + b, q − b}	{p + b, q − b}	According to Equation 7.1 we have e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block. e_IN[B5] = e_OUT[B4], since there is only 1 predecessor, i.e. B4 We have already computed **e_OUT[B4] = {**p + b, q − b**} in the previous step.** e_IN[B5] = e_OUT[B4] e_IN[B5] = {p + b, q − b} e_GEN[B5] = {ø}, since the block does not evaluate any expression. e_KILL[B5] = {ø} since There is no assignment to 'p' or 'b', which kills the expression 'p + b'. There are no assignments to 'q' or 'b', which can kill the expression 'q − b'.

				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] − e_KILL[B]), we have
				e_OUT[B5] = {p + b, q − b} U ({{ø} − {ø})
				e_OUT[B5] = {p + b, q − b} U ({ø})
				e_OUT[B5] = {p + b, q − b}

Table 7.52 *The values of e_IN and e_OUT for iteration 1 and 2*

Block #	Iteration 1		Iteration 2	
	e_IN	e_OUT	e_IN	e_OUT
0	{ø}	{p + b, q − b}	{ø}	{p + b, q − b}
1	{q − b}	{q − b}	{q − b}	{q − b}
2	{q − b}	{q − b}	{q − b}	{q − b}
3	{q − b}	{q − b}	{q − b}	{q − b}
4	{q − b}	{p + b, q − b}	{q − b}	{p + b, q − b}
5	{p + b, q − b}	{p + b, q − b}	{p + b, q − b}	{p + b, q − b}

Algorithm 7.4 summarises the computation of available expression (e_IN/e_OUT) using the iterative approach of solving data flow equations that we discussed above.

```
e_IN[B0] = ø
out[B0] = e_GEN[B0]

/* Initialize e_OUT for all blocks */
for every block B except the initial block B0 {
        e_OUT[B] = L − e_KILL[B]
}

steady_state=FALSE

while (steady_state== FALSE) {
        steady_state=TRUE
        for every block B except the initial block B0 {

                /* e_IN */
                e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block

                /* saving e_OUT to later check if we have reached steady state */
                saved_e_OUT=e_OUT[B]

                /* computing e_OUT */
                e_OUT[B] = e_GEN[B] ∪ (e_IN[B] − e_KILL[B])

                /* Checking for a steady state of e_OUT */
                if (saved_e_OUT ! = e_OUT[B]){
                         steady_state = FALSE
                }
        }
}
```

Algorithm 7.4 *Available expressions computation using the iterative approach*

Algorithm 7.4 works for the input sources involving loops and also the ones without the loops. Observe in the algorithm that the final values of e_IN and e_OUT for all the blocks in the procedure are arrived at in an iterative fashion. For an input source without any loops, the final values of e_IN and e_OUT for each block can be arrived at in the first iteration, if the computation is made in the order of flow of control, where we compute e_IN/e_OUT for a block B only after e_IN/e_OUT for all its predecessors have been computed. In such cases, the second iteration is performed only to confirm that the values of e_OUT have reached a steady state.

The iterative approach to solve the data flow equations that we saw above is used time and again in the next few sections for computing other data flow properties like reaching definition, etc.

The next section explains how the available expression information in the form of e_IN/e_OUT, computed at the block boundaries is used in global common sub-expression elimination (gcse).

Global common sub-expression elimination using available expressions The e_IN information available at the start of each block can be used for eliminating the evaluation of common sub-expression. From a conceptual standpoint, the e_IN at the start of the block represents all the expressions that have been evaluated in the procedure before reaching this block, and are available in an evaluated form. The focal point of the discussion below is on how to use the e_IN information to eliminate the common sub-expressions.

A quad '*q*' in the block B using a common sub-expression of the form '$x + y$' can be eliminated if the following conditions are satisfied.

1. The expression '$x + y$' should be available at the start of the block B as indicated by the e_IN set for the block B.
2. There should not be an assignment to either 'x' or 'y' from the start of the block B till the quad '*q*'.

We will use the input source, the locally optimised TAC and the corresponding flow graph that we first saw in Fig. 7.36 to use the above two conditions to eliminate a quad using an available expression. Recall that the Fig. 7.42 showed the same flow graph annotated with the values of e_IN, e_GEN, e_KILL and e_OUT for each of the blocks—B0 through B4 computed in the last section. It is reproduced below for convenience.

In the block B2, there is a quad (5), which uses the expression $v1 + v2$. The quad (5) in block B2 is a potential candidate for eliminating the common sub-expression, $v1 + v2$ because it satisfies the following conditions:

1. The expression $v1 + v2$ is available at the start of the block as indicated by e_IN[B2] = $\{v1 + v2\}$.
2. There are no assignments to either $v1$ or $v2$ from the start of the block B2 till the quad (5).

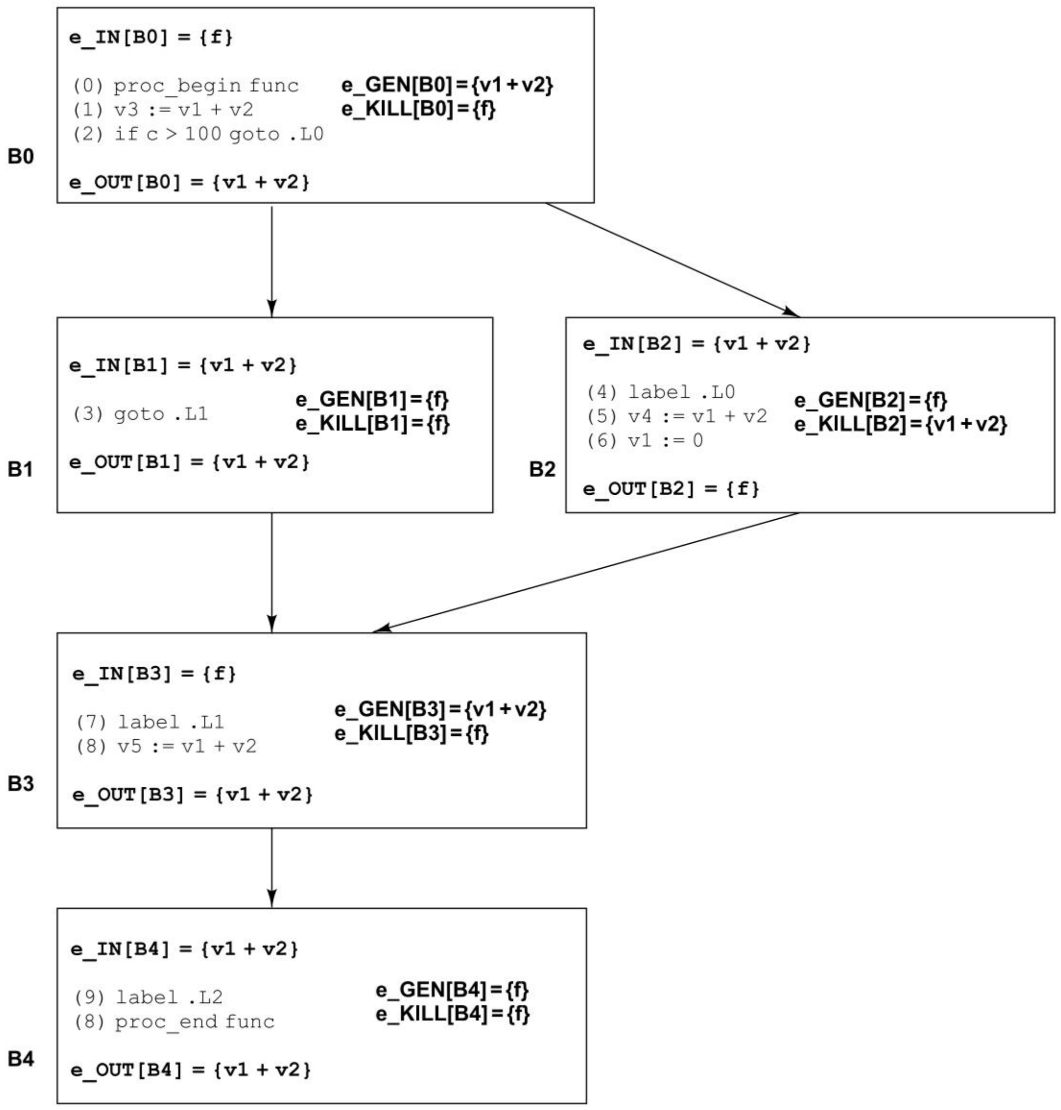
In order to eliminate the common sub-expression $v1 + v2$ in the quad (5), we first identify all the statements evaluating '$v1 + v2$' that reach quad (5). In this case it is only quad (1) in block B0 that evaluates '$v1 + v2$' and reaches B2. We create a new temporary say '*_t0*' for using as a temporary place to store the evaluated value of $v1 + v2$. We replace the quad (1) with a couple of statements allowing us to store the value of '$v1 + v2$' as shown below.

Before		After

`(1) v3: = v1 + v2`	→	`(1a) _t0 = v1 + v2`
		`(1b) v3 = _t0`

The quad (5) is then replaced to use the temporary instead of re-evaluating the expression as follows.

Before		After
`(5) v4 := v1 + v2`	→	`(5a) v4 = _t0`



The final TAC incorporating the changes just discussed is shown in Table 7.53. The final TAC shown in Table 7.53 has eliminated the re-evaluation of common sub-expression ‘$v1 + v2$’ found earlier in the quad (5).

Table 7.53 *Global common sub-expression elimination*

Locally optimised TAC	Final TAC
(0) proc_begin func	(0) proc_begin func
(1) v3 := v1 + v2	(1a) _t0 := v1 + v2
(2) if c > 100 goto .L0	(1b) v3 := _t3
(3) goto .L1	(2) if c > 100 goto .L0

`(4) label .L0`	`(3) goto .L1`
`(5) v4 := v1 + v2`	`(4) label .L0`
`(6) v1 := 0`	`(5a) v4 := _t0`
`(7) label .L1`	`(6) v1 := 0`
`(8) v5 := v1 + v2`	`(7) label .L1`
`(9) label .L2`	`(8) v5 := v1 + v2`
`(10) proc_end func`	`(9) label .L2`
	`(10) proc_end func`

The procedure we had adopted to eliminate the re-evaluation of common sub-expression '*v*1 + *v*2' at quad (5) is formalised in Algorithm 7.5. The input to the algorithm is the locally optimised TAC and the output is the TAC in which global common sub-expressions are eliminated. The algorithm shows '+' as a notional operator denoting any of the binary operators like addition (+), subtraction (–), multiplication (*), or division (/).

```
Scan all the quads in the TAC to identify the quads 'q' of the form x := y + z, that
meet the following criteria
   (a) y + z is available at the start of the block in which 'q' is located i.e. in e_IN and
   (b) There are no assignments to 'y' or 'z' in the statements before 'q' in the block.
The identified quads 'q' are added into a set 'M'.

For each quad 'q' in the set M, do the following to eliminate the common sub-expression.

   (1)  Identify all the statements evaluating 'y + z' that reach the quad 'q' and add
        them to a set 'A'.

   (2)  Create a new temporary variable '_tn' (e.g. _t0 or _t1 or _t2 etc)

   (3)  For every statement w: = y + z, in the set A, do the following
        a. _tn := y + z
        b. w := _tn

   (4)  Replace the quad 'q'  by w: = _tn
```

Algorithm 7.5 *Global common sub-expression elimination using AE*

Example 4—Global common sub-expression elimination using AE This section demonstrates the toy C compiler (mycc) performing global common sub-expression elimination using the available expression information. The available expression information was gathered by using Algorithm 7.4, explained previously. The AE information was put to use to eliminate the global common sub-expressions as explained in Algorithm 7.5.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC (b) the locally optimised TAC and (c) the TAC after global common sub-expression elimination. The dialog below shows 'mycc' taking in some sample input C sources having common sub-expressions and printing out the above information as the output.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$ g++  -DICGEN -g -Wall ic_gen.cc optimize.cc target_code_gen.cc mycc.cc
semantic_analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Sample Input C file containing common sub-expressions
$ cat -n test.cse.1.c
    1  int v1, v2, v3, v4, v5;
    2
    3  int func(int c)
    4  {
    5          v3  = v1 + v2 ;
    6
    7          if (c > 100){
    8
    9                  /* 'v1+v2' is available
   10                     here */
   11                  v4  = v1 + v2;
   12
   13                  /* CSE  Killed */
   14                  v1  = 0;
   15          }
   16
   17          /* 'v1+v2' is NOT available
   18             here */
   19          v5  = v1 + v2 ;
   20  }

# Intermediate code before and after optimization
# -O gcse for Global Common Subexpression, -v for verbosity
$ ./mycc.exe -i -O gcse -v test.cse.1.c
TAC Before optimization
 (0) proc_begin func
 (1) _t0 := v1 + v2
 (2) v3 := _t0
 (3) if c > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) _t1 := v1 + v2
 (7) v4 := _t1
 (8) v1 := 0
 (9) label .L1
(10) _t2 := v1 + v2
(11) v5 := _t2
```

```
(12) label .L2
(13) proc_end func

TAC After Local Optimization
 (0) proc_begin func
 (1) v3 : = v1 + v2
 (2) if c > 100 goto .L0
 (3) goto .L1
 (4) label .L0
 (5) v4 := v1 + v2
 (6) v1 := 0
 (7) label .L1
 (8) v5 : = v1 + v2
 (9) label .L2
(10) proc_end func

TAC After (Local and Global) optimization
 (0) proc_begin func
 (1) _t3 := v1 + v2
 (2) v3 := _t3
 (3) if c > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) v4 := _t3
 (7) v1 := 0
 (8) label .L1
 (9) v5 := v1 + v2
(10) proc_end func


# Sample Input C file containing common subexpressions
$ cat -n test.cse.2.c
     1  int v1, v2, v3, v4, v5, v6, v7;
     2
     3  int func(int c)
     4  {
     5          v3  = v1 + v2 ;
     6
     7          if (c > 100){
     8
     9                  /* 'v1+v2' is available here as CSE */
    10                  v4  = v1 + v2;
    11
    12                  /* CSE  Killed */
    13                  v1  = v3 + v4;
    14
    15                  /* computing 'v1+v2' again */
    16                  v7  = v1 + v2;
    17          }
    18
    19          /* 'v1+v2' is available here as CSE */
    20          v5  = v1 + v2 ;
    21  }

# Intermediate code before and after optimization
$ ./mycc.exe -i -O gcse -v test.cse.2.c
```

```
TAC Before optimization
 (0) proc_begin func
 (1) _t0 := v1 + v2
 (2) v3 := _t0
 (3) if c > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) _t1 := v1 + v2
 (7) v4 := _t1
 (8) _t2 := v3 + v4
 (9) v1 := _t2
(10) _t3 := v1 + v2
(11) v7 := _t3
(12) label .L1
(13) _t4 := v1 + v2
(14) v5 := _t4
(15) label .L2
(16) proc_end func

TAC After Local Optimization
 (0) proc_begin func
 (1) v3 : = v1 + v2
 (2) if c > 100 goto .L0
 (3) goto .L1
 (4) label .L0
 (5) v4 := v1 + v2
 (6) v1 := v3 + v4
 (7) v7 := v1 + v2
 (8) label .L1
 (9) v5 := v1 + v2
(10) label .L2
(11) proc_end func

TAC After (Local and Global) optimization
 (0) proc_begin func
 (1) _t5 := v1 + v2
 (2) v3 := _t5
 (3) if c > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) v4 := _t5
 (7) v1 := v3 + v4
 (8) _t5 := v1 + v2
 (9) v7 := _t5
(10) label .L1
(11) v5 := _t5
(12) proc_end func

# Sample Input C file containing common sub-expressions
$ cat -n test.cse.3.c
    1  int v1, v2, v3, v4, v5, v6;
    2
    3  int func(int c)
    4  {
    5          if (c > 100){
    6                  v3  = v1 + v2;
```

```
          }else{
                 v4  = v1 + v2 ;
          }

          /* 'v1+v2' is available here as CSE */
          v5  = v1 + v2 ;
  }

# Intermediate code before and after optimization
$ ./mycc.exe -i -O gcse -v test.cse.3.c
TAC Before optimization
 (0) proc_begin func
 (1) if c > 100 goto .L0
 (2) goto .L1
 (3) label .L0
 (4) _t0 := v1 + v2
 (5) v3 := _t0
 (6) goto .L2
 (7) label .L1
 (8) _t1 := v1 + v2
 (9) v4 := _t1
(10) label .L2
(11) _t2 := v1 + v2
(12) v5 := _t2
(13) label .L3
(14) proc_end func

TAC After Local Optimization
 (0) proc_begin func
 (1) if c > 100 goto .L0
 (2) goto .L1
 (3) label .L0
 (4) v3 := v1 + v2
 (5) goto .L2
 (6) label .L1
 (7) v4 := v1 + v2
 (8) label .L2
 (9) v5 := v1 + v2
(10) label .L3
(11) proc_end func

TAC After (Local and Global) optimization
 (0) proc_begin func
 (1) if c > 100 goto .L0
 (2) goto .L1
 (3) label .L0
 (4) _t3 := v1 + v2
 (5) v3 := _t3
 (6) goto .L2
 (7) label .L1
 (8) _t3 := v1 + v2
 (9) v4 := _t3
(10) label .L2
(11) v5 := _t3
(12) proc_end func
```

7.2.10.3 Live Variable Analysis

Introduction The live variable analysis is another commonly employed data flow analysis technique to derive a data flow property called as ***'liveness'*** from the input TAC. The liveness information helps us in performing global dead code elimination. A more important application of live variable analysis information happens during the optimised target code generation, where it is used in making decisions for retaining a variable's value in a register. We study about that later in the section on optimised target code generation.

Let's take an example to understand the idea of liveness and see how that can be used in performing dead code elimination at a global level. Consider the input source and the corresponding TAC after local optimisation shown in Fig. 7.44.

```
int func (int a, int b)
{
   int i, j, k;

   i = 45;
   j = a + b;

   /* 'i' is replaced by 45 */
   if ((a + i) > 100){
       k = a + j;
   }else{
       k = b + j;
   }
   return (k);
}
```

```
B0   (0) proc_begin func
     (1) i := 45
     (2) j:= a + b
     (3) _t1 := a + 45
     (4) if _t1 > 100 goto .L0

B1   (5) goto .L1

B2   (6) label .L0
     (7) k := a + j
     (8) goto .L2

B3   (9) label .L1
     (10) k := b + j

B4   (11) label .L2
     (12) return k
     (13) goto .L3

B5   (14) label .L3
     (15) proc_end func
```

Fig. 7.44 *Input source and the TAC after local optimisation*

Let's take a look at the TAC after local optimisation in order to get an idea on 'liveness'. In the block B0, we have an assignment *i* := 45 in quad (1). The value of '*i*', i.e. 45, has been propagated to the quad (3) during local optimisation. Observe that '*i*' is not used later in block 0 or in any of the later blocks. In contrast, the value of '*j*' computed in quad (2) of block B0 is used in other blocks, i.e. in quad (7) and quad (10). We can say that the variable '*j*' is live at the point right after quad (2) and also at the end of the block B0. The variable '*i*' is not live either at the point right after quad (1) or nor at the end of the block B0.

A variable *v* is said to be ***live*** at a point *p*, if it is used in some path in the flow graph starting *p*. The variable is considered dead, if it is not live.

The quad (1) in which we assign 45 to the variable '*i*' can be eliminated, since the variable '*i*' is dead. In general, let's say we have a quad *q*: *v* = *x* op *y*, where op is one of the operators in the TAC. If the variable '*v*' is dead and the operator op does not have any side-effects then the statement '*s*' can be eliminated. Observe that the quad (1) can be eliminated only when we have the information that the user defined variable '*i*' is not used across the blocks. This cannot be performed by analysing the TAC within a block in local optimisation. This is ***global dead code elimination***. The global dead code elimination is one of the benefits of doing live variable analysis. Table 7.54 shows the TAC after the global dead code elimination for the sample input source shown in Fig. 7.44.

Table 7.54 *TAC after global dead code elimination*

```
 (0) proc_begin func
 (1) j: = a + b
 (2) _t1: = a + 45
 (3) if _t1 > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) k := a + j
 (7) goto .L2
 (8) label .L1
 (9) k: = b + j
(10) label .L2
(11) return k
(12) goto .L3
(13) label .L3
(14) proc_end func
```

Data Flow Analysis to Compute Liveness Information in Intermediate Code In the discussion above, we understood the concept of live variable analysis and how it can be used to perform global dead code elimination. In this section we study about the algorithms that are used for computing the liveness properties in a given program.

Let's start off with some terminology.

We use the term ***live_DEFS [B]*** to represent the set of variables whose definition precedes any use in the block B. In simpler terms, if we scan the block starting from the first quad to the last quad in the block, and the earliest quad in which '*v*' is involved is a definition, then '*v*' goes in live_DEFS.

We use the term ***live_USES [B]*** to represent the set of variables whose use precedes any definition within the block B. In simpler terms, if we scan the block starting from the first quad to the last quad in the block, and the earliest quad in which '*v*' is involved is a use, then '*v*' goes in live_USES. In case of situations, where the earliest quad in the block B involving '*v*' defines and uses '*v*' (e.g. $v := v + 1$), then '*v*' goes into live_USES. This is because the quad uses the earlier value of '*v*' to redefine '*v*'.

Observe that live_DEFS and live_USES are mutually exclusive sets, since we can either find a definition of a variable '*v*' earlier or the usage of a variable '*v*' earlier.

A simple algorithm to calculate live_USES and live_DEFS for a block is given below.

```
for each quad  res := arg1 op arg2 in the Block
{
        if (arg1 is valid for the operator){
             if (arg1 is not in live_DEFS){
                  /* Use has preceded any definition */
                  insert (live_USES,arg1)
             }
        }

        if (arg2 is valid for the operator){
             if (arg2 is not in live_DEFS){
                /* Use has preceded any definition */
                insert (live_USES,arg1)
            }
        }

        if (res is valid for the operator){
            if (res is not in live_USES){
                /* Definition has preceded any use */
                insert (live_DEFS,arg1)
            }
        }
}
```

Algorithm 7.6 *live_USES and live_DEFS*

The following table shows how live_DEFS and live_USES sets are computed for the TAC after local optimisation shown in Fig. 7.44.

Table 7.55 *The calculation of live_DEFS and live_USES*

#	TAC after local optimisation	live_DEFS and live_USES	Explanation

B0	(0) proc_begin func
	(1) i := 45
	(2) j := a + b
	(3) _t1 := a + 45
	(4) if _t1 > 100 goto .L0
		live_DEFS[B0] = {i, j, _t1}
		live_USES[B0] = {a, b}
			The variable 'i' is defined in quad (1) and is not used in the block before the quad (1).
			
			The variable 'j' is defined in quad (2) and is not used in this block before quad(2).
			
			The variable '_t1' is defined in quad (3) and is not used before quad (3).
			
			All of these qualify for live_DEFS.
			
			The variables 'a' and 'b' are used in quad (2). There is no definition of 'a' or 'b' preceding the quad (2) in the block B0. Hence, they qualify to be part of live_USES.

B1	(5) goto .L1	live_USES[B1] = {}
		live_DEFS[B1] = {}
			No variable is defined or used in this block

B2	(6) label .L0
	(7) k := a + j
	(8) goto .L2
		live_DEFS[B2] = {k}
		live_USES[B2] = {a, j}
			The variable 'k' is defined in quad (7) and is not used in the block before the quad (7)
			
			The variables 'a' and 'j' are used in quad (7) and there is no definition for any of them preceding quad (7) in this block.

B3	(9) label .L1
	(10) k := b + j
		live_DEFS[B3] = {k}
		live_USES[B3] = {b, j}
			The variable 'k' is defined in quad (10) and is not used in the block before the quad (10)
			
			The variables 'b' and 'j' are used in quad (10) and there is no definition for any of them preceding quad (10) in this block.

B4	(11) label .L2
	(12) return k
	(13) goto .L3
		live_DEFS[B4] = {}
		live_USES[B4] = {k}
			No variable is defined in this block
			
			The variable 'k' is used in quad (12) and there is no definition for 'k' preceding quad (12) in this block.

B5	(14) label .L3
	(15) proc_end func
		live_USES[B5] = {}
		live_DEFS[B5] = {}
			No variable is defined or used in this block

As seen earlier during the discussion on AE, the data flow properties are computed using equations known as ***data flow equations***. By using the data flow equations, one can compute the values of data flow property like liveness of variables at the block boundaries. The value of data flow property is then extrapolated to each quad within the block by using its value at the block boundary.

The data flow equations for live variable analysis express the relationship between the ***live_IN[B]***, the set of all the variables that are live before reaching the beginning of block 'B' to ***live_OUT[B]*** the set of variables that are live at the end of block 'B'. The data flow equations for live variable analysis are as follows.

live_IN[B] = live_USES[B] ∪ (live_OUT[B] – live_DEFS[B]) live_OUT[B] = live_IN[S] ∪ live_IN[S2] ∪ live_IN[S3] ….for all the successors S1, S2, S3….Sn of the block	Equation 7.3

We will continue the example in Table 7.55 to calculate live_IN and live_OUT for each of the blocks by using the data flow equations given by Equation 7.3. Table 7.56 shows the calculation of the sets live_IN and live_OUT for all the blocks B0 through B5 using the data flow equations. The calculations for live_IN and live_OUT have been done starting from the block B5 and ending at block B0, since the data flow equation involves all the successors (live_OUT[B] = ∪ live_IN[S] for all the successors S of the block). We would want the live_IN of the successors to be populated before we calculate the live_OUT of a block. Observe that the computation of live variable analysis involves working backward from the last block to the starting block.

Table 7.56 *live_IN and live_OUT calculations*

#	TAC after local optimisation	live_DEFS and live_USES	live_IN and live_OUT	Explanation

B5	(14) label .L3
	(15) proc_end func
		live_USES[B5] = {}
		live_DEFS[B5] = {}
			live_OUT[B5] = {}
			live_IN[B5] = {}
				live_OUT[B5] = {}, since there is NO succesor of B5. Hence live_OUT[B5] = {}

				live_IN[B5] = live_USES[B5] ∪ (live_OUT[B5] – live_DEFS[B5])

B4	(11) label .L2
	(12) return k
	(13) goto .L3
		live_DEFS[B4] = {}
		live_USES[B4] = {k}
			live_OUT[B4] = {}
			live_IN[B4]={k}
				live_OUT[B4] = ∪ live_IN[B5], since B5 is the only succesor of B4. Hence live_OUT[B4] = {}
				
				live_IN[B4] = live_USES[B4] ∪ (live_OUT[B4] – live_DEFS[B4])
				
				live_IN [B4] = {k}

B3	(9) label .L1
	(10) k := b + j
		live_DEFS[B3] = {k}
		live_USES[B3] = {b, j}
			live_OUT[B3] = {k}
			
			live_IN[B3] = {b, j}
				live_OUT[B3] = ∪ live_IN[B4], since B4 is the only succesor of B3. Hence live_OUT[B3] = {k}
				
				live_IN[B3] = live_USES[B3] ∪ (live_OUT[B3] – live_DEFS[B3])
				
				(live_IN[B3] = {b,j} ∪ {[k] - {k}}
				live_IN[B3] = {b,j}

B2	(6) label .L0
	(7) k := a + j
	(8) goto .L2
		live_DEFS[B2] = {k}
		live_USES[B2] = {a, j}
			live_OUT[B2] = {k}
			live_IN[B2] = {a, j}
				live_OUT[B2] = ∪ live_IN[B4], since B4 is the only successor of B2. Hence live_OUT[B2] = {k}
				
				live_IN[B2] = live_USES[B2] ∪ (live_OUT[B2] – live_DEFS[B2])
				
				live_IN[B2] = {a,j) ∪ {[k}-{k}]
				
				live_IN [B2] = {a,j}

B1	(5) goto .L1	live_USES[B1] = {}
		live_DEFS[B1] = {}
			live_OUT[B1] = {b, j}
			live_IN[B2] = {b, j}
				live_OUT[B1] = ∪ live_IN[B3], since B3 is the only successor of B1. Hence live_OUT[B1] = {b, j}
				
				live_IN[B1] = live_USES[B1] ∪

				(live_OUT[B1] – live_DEFS[B1]) live_IN[B1}={}∪(b,j) live-IN[B2] = {b,j}

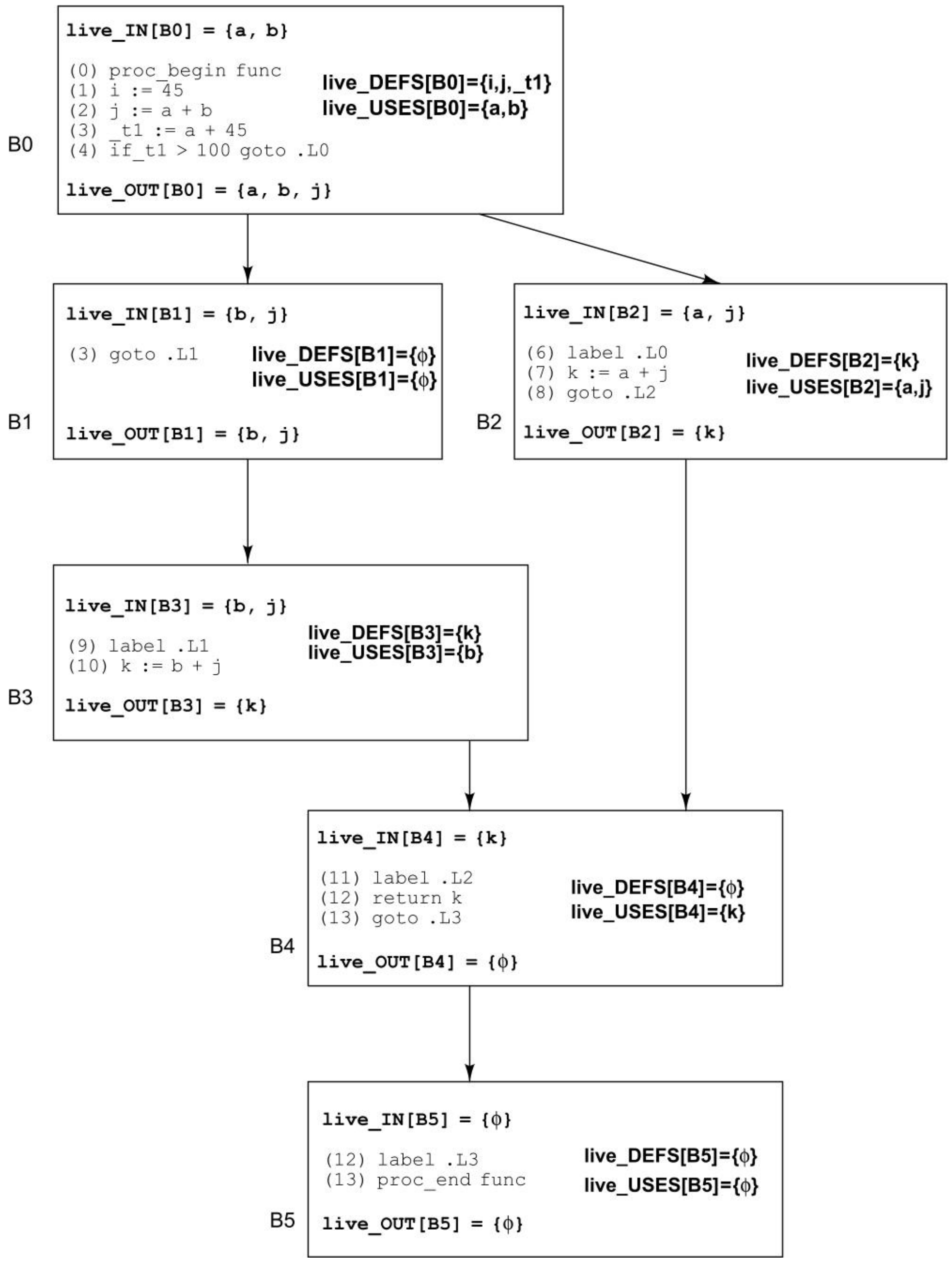
B0	(0) proc_begin func
	(1) i: = 45
	(2) j: = a + b
	(3) _t1: = a + 45
	(4) if _t1 > 100
	goto .L0
		live_DEFS[B0] = {i, j, _t1}
		live_USES[B0] = {a, b}
			live_OUT[B0] = {a, b, j}
			live_IN[B2] = {a, b}
				live_OUT[B0] =
				live_IN[B1] ∪
				live_IN[B2], since
				B1 and B2 are
				successors of B0.
				Hence live_OUT[B0]
				= {b, j} ∪ {a, j}
				
				live_OUT[B0] = {a,
				b, j}
				
				live_IN[B0] =
				live_USES[B0] ∪
				(live_OUT[B0]
				– live_DEFS[B0])
				
				live_IN[B0] = {a,
				b} ∪ {{a, b, j}
				– {i, j, _t1}}
				
				live_IN[B0] = {a,
				b} ∪ {a, b}
				
				live_IN[B2] = {a, b}

Figure 7.45 shows the flow graph annotated with the values of live_IN, live_DEFS, live_USES and live_OUT for each of the blocks—B0 through B5.

In this example, we have been able to compute the live variable information in the form of live_IN and live_OUT by using the Equation 7.3 for all the blocks in the program. The values of live_IN and live_OUT for each of the blocks was calculated by proceeding block after block in the reverse order as the data flow, i.e. B5, followed B4, B3, B2, B1 and B0. We had to strictly follow the reverse order of flow of control while computing the live_OUT because the live_OUT[B] = U live_IN[S] for all the successors S of the block. We could not have computed live_OUT[B], even if the live_IN of one of the successors were not yet computed. We had seen earlier during the discussion on the available expressions that the cyclic dependency between the nodes of the flow graph makes it impossible to calculate 'IN/OUT' at some of the nodes in the cases of input source having loops. We had overcome the issue for available expressions in the last section by having an ***iterative approach*** to solving the data flow equations. We do the same for live variable analysis also.

In the iterative approach of solving data flow equations for live variable analysis, an initial value for live_IN[B] for every block B is assumed. This helps us compute the live_OUT[B], where live_OUT[B] = ∪ live_IN[S] for all the successors S of the block, even in cases where live_IN[S] is not yet computed. The initial value of live_IN[B] for every block B that is assigned before the start of the first iteration is = {ø}. Observe that this initial value is the minimum value that live_IN can take.

Algorithm 7.7 summarises the computation of live variable analysis information (live_IN / live_OUT) using the iterative approach of solving data flow equations that we discussed above.



```
live_IN[B0] = {a, b}

(0) proc_begin func
(1) i := 45
(2) j := a + b
(3) _t1 := a + 45
(4) if_t1 > 100 goto .L0

live_OUT[B0] = {a, b, j}

live_DEFS[B0]={i,j,_t1}
live_USES[B0]={a,b}
```

```
live_IN[B1] = {b, j}

(3) goto .L1

live_OUT[B1] = {b, j}

live_DEFS[B1]={ϕ}
live_USES[B1]={ϕ}
```

```
live_IN[B2] = {a, j}

(6) label .L0
(7) k := a + j
(8) goto .L2

live_OUT[B2] = {k}

live_DEFS[B2]={k}
live_USES[B2]={a,j}
```

```
live_IN[B3] = {b, j}

(9) label .L1
(10) k := b + j

live_OUT[B3] = {k}

live_DEFS[B3]={k}
live_USES[B3]={b}
```

```
live_IN[B4] = {k}

(11) label .L2
(12) return k
(13) goto .L3

live_OUT[B4] = {ϕ}

live_DEFS[B4]={ϕ}
live_USES[B4]={k}
```

```
live_IN[B5] = {ϕ}

(12) label .L3
(13) proc_end func

live_OUT[B5] = {ϕ}

live_DEFS[B5]={ϕ}
live_USES[B5]={ϕ}
```

Fig. 7.45 *Flow graph annotated with live variable analysis information*

Algorithm 7.7 works for the input sources involving loops and also for the ones without the loops. Observe in the algorithm that the final values of live_IN and live_OUT for all the blocks in the procedure are arrived at in an iterative fashion. For an input source without any loops, the final values of live_IN and live_OUT for each block can be arrived at in the first iteration, if the computation is made in the order of reverse of flow of control, where we compute live_IN / live_OUT for a block B only after live_IN / live_OUT for all its successors have been computed. In such cases, the second iteration is performed only to confirm that the values of live_OUT have reached a steady state.

```
/* Initialise live_IN for all blocks */
for every block B {
        live_IN[B] = {ø}
}

steady_state = FALSE

while (steady_state = = FALSE) {
        steady_state = TRUE
        for every block B {

             /* live_OUT */
             live_OUT[B] = U live_IN[S] for all the Successors S of the block

             /* saving live_OUT to later check if we have reached steady state */
             saved_live_OUT = live_OUT[B]

             /* computing live_OUT */
             live_OUT[B] = live_GEN[B] ∪ (live_IN[B] - live_KILL[B])

             /* Checking for a steady state of live_OUT */
             if (saved_live_OUT ! = live_OUT[B]){
                  steady_state = FALSE
             }
        }
}
```

Algorithm 7.7 *Live variable analysis information computation using the iterative approach*

Dead Code Elimination using Live Variable Analysis Information The live_OUT information representing the live variables at the end of the block can be extrapolated to calculate the live variables at the beginning and the end of each quad. This information is then used to eliminate dead code.

The live_OUT information available at the end of the block is extrapolated to each quad level by working backwards from the last quad of the block. The live_OUT of the last quad in the block is the same as the live_OUT of the block. Using the live_OUT of the last quad, we can calculate live_IN for the same using the relationship

live_IN $[q_n]$ = live_USES $[q_n] \cup$ (live_OUT $[q_n]$ – live_DEFS $[q_n]$), where 'n' is the quad number.

And then, we can use the live_IN of the current quad as the live_OUT of the previous quad.

live_OUT $[q_{n-1}]$ = live_IN $[q_n]$

The process repeats itself till the first quad in the block.

Let's take the example of block B0 shown in Fig. 7.45 and calculate the live_OUT information for each of the quad. The quads in block B0 along with the live_OUT[B0] is shown below again for convenience.

B0	(0) proc_begin func (1) i := 45 (2) j := a + b (3) _t1 := a + 45 (4) if _t1 > 100 goto .L0	live_OUT[B0] = {a,b,j}

The calculation of live_OUT at the end of each quad is given along with explanations in Table 7.57.

Table 7.57 *Calculating the live_OUT information for each quad*

quad #	quad	live_OUT	Explanation	live_IN

4	if _t1 > 100 goto .L0	{a,b,j}	The live_OUT of the last quad in a block is the same as live_OUT of the block.
			
			live_OUT [q4] = {a,b,j}
			
			The instruction if _t1 > 100 goto .L0 does not define any variable, but uses the variable '_t1' .
			live_DEFS[q4] = { }
			live_USES[q4] = { _t1 }
			
			From the equation
			
			live_IN [q4] = live_USES [q4] ∪ (live_OUT [q4] − live_DEFS [q4])
			
			live_IN [q4] = { _t1 } U { {a,b,j} − { } }
			
			live_IN [q4] = { _t1,a,b,j }
				{_t1,a,b,j}

3	_t1 := a + 45	{_t1,a,b,j}	live_OUT[q3] = live_IN [q4]
			
			Hence
			live_OUT[q3] = { _t1,a,b,j }
			
			The instruction '_t1 := a + 45' defines '_t1', and uses the variable 'a' .
			live_DEFS [q3] = { _t1 }
			live_USES [q3] = { a }
			
			From the equation
			
			live_IN [q3] = live_USES [q3] ∪ (live_OUT [q3] − live_DEFS [q3])
			
			live_IN [q3] = { a } U { {_t1,a,b,j} − { _t1 } }
			
			live_IN [q3] = { a,b,j }
				{ a,b,j }

2	j := a + b	{ a,b,j }	live_OUT[q2] = live_IN [q3]
			
			Hence
			live_OUT[q2] = { a,b,j }
			
			The instruction 'j := a + b' defines 'j', and uses the variable 'a' and 'b' .
			live_DEFS [q2] = { j }
			live_USES [q2] = { a, b }
			
			From the equation
			
			live_IN [q2] = live_USES [q2] U (live_OUT [q2] – live_DEFS [q2])
			
			live_IN [q2] = { a,b } U { { a,b,j} – { j } }
			
			live_IN [q2] = { a,b }
				{ a,b }

1	i := 45	{ a,b }	live_OUT[q1] = live_IN [q2] Hence live_OUT[q1] = { a,b } The instruction 'i := 45' defines 'i', and uses no variables. live_DEFS [q1] = { i } live_USES [q1] = { } From the equation live_IN [q1] = live_USES [q1] ∪ (live_OUT [q1] – live_DEFS [q1]) live_IN [q1] = { } U { { a,b} – { i } } live_IN [q1] = { a,b }	{ a,b }

The procedure we just used for calculating the live_OUT information for each quad is formalised in Algorithm 7.8.

```
B is the current Block
live_OUT[B] is already computed using iterative approach explained earlier
```

```
out = live_OUT[B]

for each quad 'q' of the form x := y op z in the block starting with the last quad in
the block and going towards the first quad

            live_out[q] = out;
```

```
        in = def = use = { ϕ }

        use.insert(y);
        use.insert(z);

        if(use does not contain 'x' ){ /* handling m = m + c type of statements */
            def.insert(x);
        }

        in = ( out - def ) ∪ use

        /* for the next iteration */
        out=in
```

Algorithm 7.8 *live_out calculation for each quad*

We shall now see how the live out information at each quad can be used in the global dead code elimination. We use the block B0 in the above example for which we had computed live out information at each quad level. We summarise the live_OUT values at the end of each quad for the block B0 below for ease of reference.

#	Quad	live_OUT
0	proc_begin func	{ a, b}
1	i := 45	{ a b }
2	j := a + b	{ a b j }
3	_t1 := a + 45	{ a b j _t1 }
4	if _t1 > 100 goto .L0	{ a b j }

We turn our attention to the quad #1, given by *i* := 45. Observe that the variable '*i*' is not a member of the live_OUT set at the quad #1. In other words the variable '*i*' is dead at this point. Now, the quad #1 in which we define the variable '*i*' can be eliminated, since the variable '*i*' is dead at this point. In general, for a quad *q*: *v* = *x* op *y*, if the variable '*v*' is dead (i.e. not in live_OUT[q]) and the operator op does not have any side-effects (like say CALL does) then the quad '*q*' can be eliminated. This is how global dead code elimination can be performed using the live variable Information. The TAC after global dead code elimination is shown in Table 7.58.

Table 7.58 *Global dead code elimination*

TAC after local optimisation		TAC after global dead code elimination
(0) proc_begin func (1) i := 45 (2) j := a + b (3) _t1 := a + 45		(0) proc_begin func (1) j := a + b (2) _t1 := a + 45 (3) if _t1 > 100 goto .L0

```
(4)  if _t1 > 100 goto .L0          (4)  goto .L1
 (5) goto .L1                       (5)  label .L0
 (6) label .L0                 →    (6)  k := a + j
 (7) k := a + j                     (7)  goto .L2
 (8) goto .L2                       (8)  label .L1
 (9) label .L1                      (9)  k := b + j
(10) k := b + j                    (10)  label .L2
(11) label .L2                     (11)  return k
(12) return k                      (12)  goto .L3
(13) goto .L3                      (13)  label .L3
(14) label .L3                     (14)  proc_end func
(15) proc_end func
```

Algorithm 7.9 formalises the idea of global dead code elimination using the live out information.

```
for each quad 'q' of the form x:= y op z in the block
{
    if( live_out[q] does not contain x ) { /* x is dead */
       eliminate the quad q
    }
}
```

Algorithm 7.9 *Global dead code elimination using live_out information*

The live variable analysis information can be used at a DAG level in the local optimisation to eliminate the dead stores. For example, consider the DAG for the block B0 shown in Fig. 7.46.

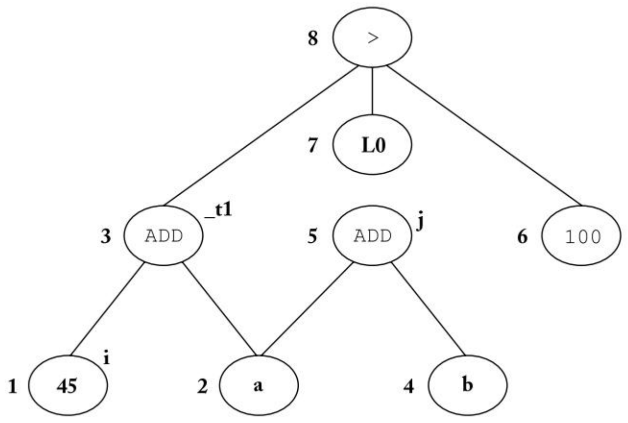


Fig. 7.46 *DAG for block 0*

At the node 1, according to the Algorithm 7.3, we generate the assignment statement *i* := 45, since the attached identifier '*i*' is an user defined variable. We can modify the algorithm to generate an assignment statement only, if the identifier in the attached list is needed outside the block, i.e. in the live_OUT of the block. Since '*i*' is not part of the live_OUT set for block B0, the assignment statement *i* := 45 is not generated, when the quads are regenerated from the DAG. In this way, dead code elimination is performed at a DAG level using live variable information in the form of live_OUT set for the block.

Example 5—Global dead code elimination using live variable analysis This section demonstrates the toy C compiler (mycc) performing global dead code elimination using the live variable analysis information. The live variable analysis information was gathered by using the algorithms explained in the preceding section. The live_OUT information was put to use to eliminate the dead code as explained in Algorithm 7.9.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC (b) the locally optimised TAC and (c) the TAC after global dead code elimination. The dialog below shows ‘mycc’ taking in some sample input C sources having dead code and printing out the above information as the output.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building ‘mycc’ - A Toy Compiler for C Language
$ g++  -DICGEN -g -Wall ic_gen.cc optimize.cc target_code_gen.cc mycc.cc semantic_
analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Sample Input C file containing dead code that can be eliminated
$ cat -n test.dce.1.c
    1  int func(int a,int b)
    2  {
    3      int i,j,k;
    4
    5      i=45; /* dead code after local opt*/
    6      j=a+b;
    7
    8      /* ‘i’ is replaced by 45 */
    9      if((a+i) > 100 ){
   10          k=a+j;
   11      }else{
   12          k=b+j;
   13      }
   14      return(k);
   15  }
   16

# Intermediate code before and after optimization
# -O gdce for Global Dead Code Elimination, -v for verbosity
$ ./mycc.exe -i -O gdce -v test.dce.1.c
TAC Before optimisation
 (0) proc_begin func
 (1) i := 45
 (2) _t0 := a + b
 (3) j := _t0
 (4) _t1 := a + i
 (5) if _t1 > 100 goto .L0
 (6) goto .L1
 (7) label .L0
 (8) _t2 := a + j
 (9) k := _t2
(10) goto .L2
(11) label .L1
(12) _t3 := b + j
(13) k := _t3
(14) label .L2
(15) return k
(16) goto .L3
(17) label .L3
(18) proc_end func

TAC After Local Optimization
 (0) proc_begin func
 (1) i := 45
 (2) j := a + b
 (3) _t1 := a + 45
 (4) if _t1 > 100 goto .L0
 (5) goto .L1
 (6) label .L0
 (7) k := a + j
 (8) goto .L2
 (9) label .L1
(10) k := b + j
(11) label .L2
(12) return k
(13) goto .L3
(14) label .L3
(15) proc_end func

TAC After (Local and Global) optimization
 (0) proc_begin func
 (1) j := a + b
 (2) _t1 := a + 45
 (3) if _t1 > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) k := a + j
 (7) goto .L2
 (8) label .L1
 (9) k := b + j
(10) label .L2
(11) return k
(12) goto .L3
(13) label .L3
(14) proc_end func

# Sample Input C file containing dead code that can be eliminated
$ cat -n test.dce.2.c
    1
    2  int func(int a, int b)
```

```
{
     int i,j;

     i= a * b; /* dead store */

     if(a > 100 ){
          i=a+b; /* i redefined */
          j=j+i;
     }else{
          i=a-b; /* i redefined */
          j=b*a;
          j = j + a;
     }
     return(j+i);


}

```

```
# Intermediate code before and after optimization
$ ./mycc.exe -i -O gdce -v test.dce.2.c
TAC Before optimization
 (0) proc_begin func
 (1) _t0 := a * b
 (2) i := _t0
 (3) if a > 100 goto .L0
 (4) goto .L1
 (5) label .L0
 (6) _t1 := a + b
 (7) i := _t1
 (8) _t2 := j + i
 (9) j := _t2
(10) goto .L2
(11) label .L1
(12) _t3 := a - b
(13) i := _t3
(14) _t4 := b * a
(15) j := _t4
(16) _t5 := j + a
(17) j := _t5
(18) label .L2
(19) _t6 := j + i
(20) return _t6
(21) goto .L3
(22) label .L3
(23) proc_end func

TAC After Local Optimization
 (0) proc_begin func
 (1) i := a * b
 (2) if a > 100 goto .L0
 (3) goto .L1
 (4) label .L0
 (5) i := a + b
 (6) j := j + i
 (7) goto .L2
 (8) label .L1
 (9) i := a - b
(10) _t4 := b * a
(11) j := _t4 + a
(12) label .L2
(13) _t6 := j + i
(14) return _t6
(15) goto .L3
(16) label .L3
(17) proc_end func

TAC After (Local and Global) optimization
 (0) proc_begin func
 (1) if a > 100 goto .L0
 (2) goto .L1
 (3) label .L0
 (4) i := a + b
 (5) j := j + i
 (6) goto .L2
 (7) label .L1
 (8) i := a - b
 (9) _t4 := b * a
(10) j := _t4 + a
(11) label .L2
(12) _t6 := j + i
(13) return _t6
(14) goto .L3
(15) label .L3
(16) proc_end func
```

7.2.10.4 Loops—An Introduction We have studied about the data flow analysis to compute the ‘available expression’ and ‘liveness’ properties of a given TAC. We saw that these properties were used to perform global common sub-expression elimination and global dead code elimination in the program respectively. The next data flow property that we study about is called as ‘reaching definitions’. We use the reaching definitions property to perform optimisations in a loop.

Before we start studying about the reaching definitions, we take a short diversion to understand the basic concepts and ideas with regard to loops. In this section, we examine the concepts and algorithms to:

(a) Detect the presence of a loop, given the intermediate code.
(b) Identify the basic blocks in the intermediate code that constitute a loop.

The ideas and the algorithms presented here are pre-requisites to the study of loop optimisation. We study about one of the loop optimisations later by using reaching definitions.

Detection of Loop

The loops in programs are detected during the data flow analysis by using a concept called as ***‘domination’*** in a flow graph.

A node ‘*d*’ of a flow graph dominates node ‘*n*’, if every path from the initial node to ‘*n*’ goes through ‘*d*’. It is represented as ***d dom n***. By definition, each node dominates itself.

Consider the input source, TAC after local optimisation and the corresponding flow graph in Fig. 7.47 to understand the idea of domination.

In Fig. 7.47, the initial node of the flow graph is B0. We can see from the flow graph that in order to reach block B4 from the initial node B0, it is mandatory to go through block B1. It is impossible to reach

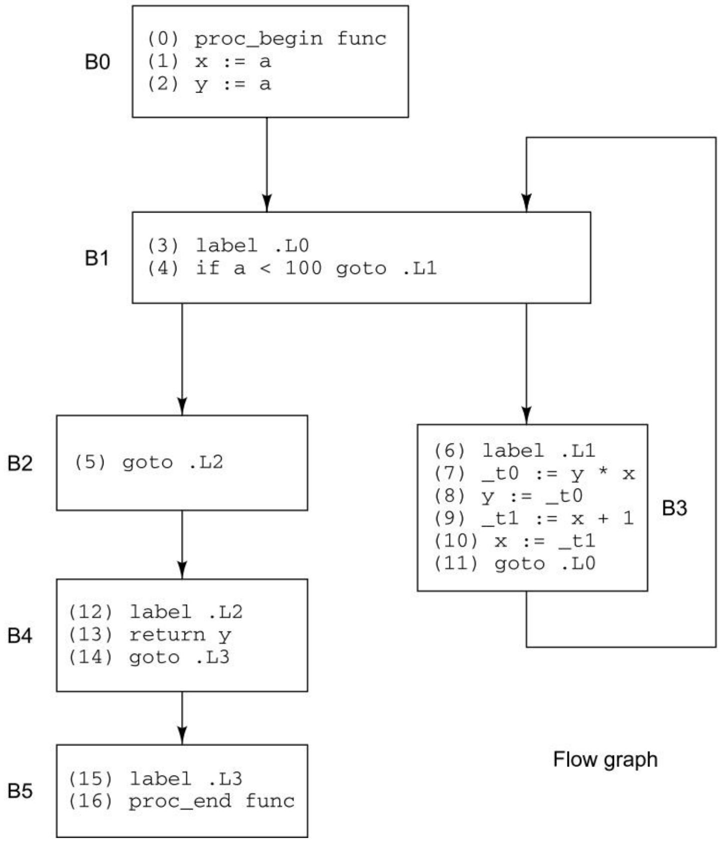
B4 from initial node B0 without traversing B1. In other words, B1 dominates B4 (B1 dom B4). In the same flow graph, in order to reach block B4 from the initial node B0, it is not necessary that the control needs to always pass through the block B3. Hence, we can say that B3 does not dominate B4.

```

int func(int a)
{

  int x, y;

  x = a;
  y = a;

  while (a < 100){
    y = y * x;
    x = x + 1;
  }

   return (y);
 }

```

Input source

```
(0) proc_begin func
(1) x := a
(2) y := a

(3) label .L0
(4) if a < 100 goto .L1

(5) goto .L2

(6) label .L1
(7) _t0 := 7 * x
(8) y := _t0
(9) _t1 := x + 1
(10) x := _t1
(11) goto .L0

(12) label .L2
(13) return y
(14) goto .L3

(15) label .L3
(16) proc_end func
```

TAC after local optimisation



```
B0:
(0) proc_begin func
(1) x := a
(2) y := a

B1:
(3) label .L0
(4) if a < 100 goto .L1

B2:
(5) goto .L2

B3:
(6) label .L1
(7) _t0 := y * x
(8) y := _t0
(9) _t1 := x + 1
(10) x := _t1
(11) goto .L0

B4:
(12) label .L2
(13) return y
(14) goto .L3

B5:
(15) label .L3
(16) proc_end func
```

Flow graph

Fig. 7.47 *Input source, TAC after local optimisation and flow graph*

We use the term ***dominators*** to represent the set of nodes that dominate a particular node. For example, in Fig. 7.47, we can see that B0, B1 and B2 dominate B2. Thus, the dominators [B2] = {B0, B1, B2} or simply dominators [2] = {0, 1, 2}.

Table 7.59 shows the dominators for each of the nodes in the flow graph. Observe that the dominators set is never empty because (a) each node dominates itself and (b) the initial node also has to be part of the dominators, since the definition of dominations relates to path starting from initial node.

Table 7.59 *Dominators*

```
dominators [ 0 ] = { 0 }
dominators [ 1 ] = { 0, 1 }
dominators [ 2 ] = { 0, 1, 2 }
dominators [ 3 ] = { 0, 1, 3 }
dominators [ 4 ] = { 0, 1, 2, 4 }
dominators [ 5 ] = { 0, 1, 2, 4, 5 }
```

An edge in a flow graph represents a possible flow of control. For example, in Fig. 7.47 the edge B0 to B1 written as 0 → 1 represents a possible flow of control. The edges in the flow graph of Fig. 7.47 are 0 → 1, 1 → 2, 1 → 3, 3 → 1, 2 → 4 and 4 → 5. For an edge in a flow graph denoted by $a \to b$, the node '*b*' is called as the ***head*** and the node '*a*' is called as the ***tail***. It is normal to find that the dominators[head] containing the tail, since there is an edge from the tail to head. However, there are some edges in which dominators[tail] contain the head. These are called the ***back edges***. The presence of a back edge indicates the existence of a loop in a flow graph. Table 7.60 shows each of the edges in the flow graph along with the dominators for the head and tail of those edges. We can see from the table, that the back edge 3 → 1 has been detected by the presence of head node in the dominators[tail].

Table 7.60 *Edges and dominators for head and tail*

Edge	Head	Tail	dominators[head]	dominators[tail]	Remarks
0→1	1	0	{ 0, 1 }	{ 0 }	
1→2	2	1	{ 0, 1, 2 }	{ 0, 1 }	
1→3	3	1	{ 0, 1, 3 }	{ 0, 1 }	
3→1	1	3	{ 0, 1 }	{ 0, $\underline{\mathbf{1}}$, 3 }	Back Edge
2→4	4	2	{ 0, 1, 2, 4 }	{ 0, 1, 2 }	
4→5	5	4	{ 0, 1, 2, 4, 5 }	{ 0, 1, 2, 4 }	

Let's take another flow graph shown in Fig. 7.48 as an example to firm up the ideas on domination. In Fig. 7.48, by visual inspection, we can notice that (B2, B4, B6) form a loop and (B1, B3) form another loop.

Table 7.61 shows the dominators for each of the nodes in the flow graph.

Table 7.61 Dominators

dominators [0]	=	{ 0 }
dominators [1]	=	{ 0 , 1 }
dominators [2]	=	{ 0 , 2 }
dominators [3]	=	{ 0 , 1 , 3 }
dominators [4]	=	{ 0 , 2 , 4 }
dominators [5]	=	{ 0, 1 , 3 , 5 }
dominators [6]	=	{ 0 , 2 , 4 , 6 }

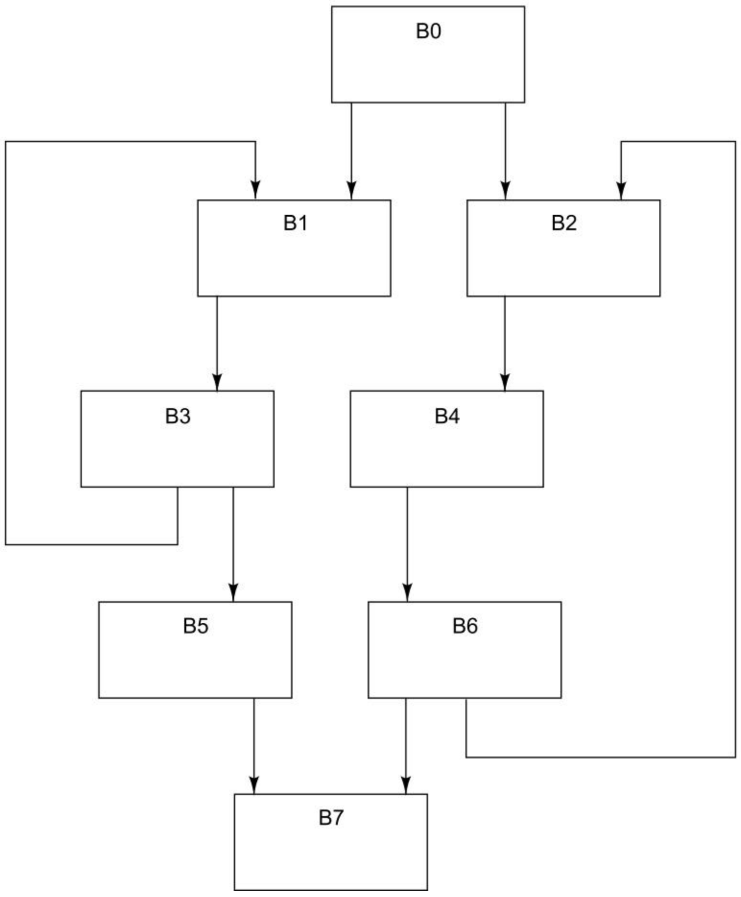


Fig. 7.48 *Flow graph*

Table 7.62 shows each of the edges in the flow graph along with the dominators for the head and tail of those edges. We can see from the table, that the back edges 3 → 1 and 6 → 2 have been detected by the presence of head node in the dominators[tail]. This identifies both the loops.

In a loop, the entry of the loop dominates all the nodes in the loop. In Fig. 7.48, {B6, B2, B4} form a loop L1 and {B3, B1} form another loop—L2. The entry of the loop L1 is B2. The other nodes in the loop L1 are B4 and B6. We can see that B2 is present in the dominator set of B4 and B6, i.e. in dominators[B4] and dominators[B6]. The entry of the loop is also called as the ***header*** of the loop. The Loop L1 can be exited from the basic block B6. It is called as ***loop exit block***. The block B3 is the loop exit block for the loop L2. It is possible to have multiple exit blocks in a loop.

A loop L can be formally described as a set of nodes $\{n_1, n_2, n_3 \ldots n_k\}$ in the flow graph in which there is path from any node n_i to n_j, via other nodes that are also part of the loop L. The header node of the loop dominates all the nodes in the loop.

In some of the loop optimisation techniques like, say, code motion, it is required to move several quads from within the loop to outside of the loop. In the optimised code, these quads would typically need to be executed before entering the loop.

Table 7.62 *Edges and dominators for head and tail*

Edge	Head	Tail	dominators[head]	dominators[tail]	Remarks
$0 \rightarrow 1$	1	0	{0, 1}	{0}	
$0 \rightarrow 2$	2	0	{0, 2}	{0}	
$1 \rightarrow 3$	3	1	{0, 1, 3}	{0, 1}	
$3 \rightarrow 1$	1	3	{0, 1}	{0, **1**, 3}	Back edge
$3 \rightarrow 5$	5	3	{0, 1, 3, 5}	{0, 1, 3}	
$5 \rightarrow 7$	7	5	{0, 7}	{0 1, 3, 5}	
$2 \rightarrow 4$	4	2	{0, 2, 4}	{0, 2}	
$6 \rightarrow 2$	2	6	{0, 2}	{0, **2**, 4, 6}	Back edge
$4 \rightarrow 6$	6	4	{0, 2, 4, 6}	{0, 2, 4}	
$6 \rightarrow 7$	7	6	{0, 7}	{0, 2, 4, 6}	

A ***pre-header*** block serves as a placeholder for the quads that need to be executed just before entering the loop. The pre-header is a basic block introduced during the loop optimisation to hold the quads that are moved from within the loop. It is a predecessor to the header block. Figure 7.49 illustrates the idea of a pre-header block.

Figure 7.49 (A) shows a flow graph with of a loop L consisting of nodes {B3, B4, B5} with B3 as the header. Imagine, during the data flow analysis we find that there are quads in say B4, which can be moved out of the loop. The optimiser introduces a pre-header block B7, which is a predecessor to the header of the loop B3 and moves the quads into it. The block B7 is a successor to the blocks B1 and B2, which were earlier flowing to B3.

Identifying the Basic Blocks forming a Loop

In data flow analysis the presence of a back edge indicates a loop in the program. In order to make optimisations in the loop, it is required to know the nodes that constitute a loop in the flow graph. For example, in Fig. 7.48, the loop L1 constitutes of {B6, B2, B4} and loop L2 constitutes of {B1, B3}.

In data flow analysis, the constituent blocks in a loop are usually calculated from the back edge. Given a back edge, $n1 \rightarrow n2$, A ***natural loop*** is $n1$, $n2$ and the set of nodes that can reach '$n1$' without going through '$n2$'. We can observe from Fig. 7.48 that for the back edge $6 \rightarrow 2$, the set of nodes that can reach 6 without going through 2 is B4. In other words, there is a path from B4 to B6 without going through B2. By this definition {B6, B2, B4} constitute a natural loop.

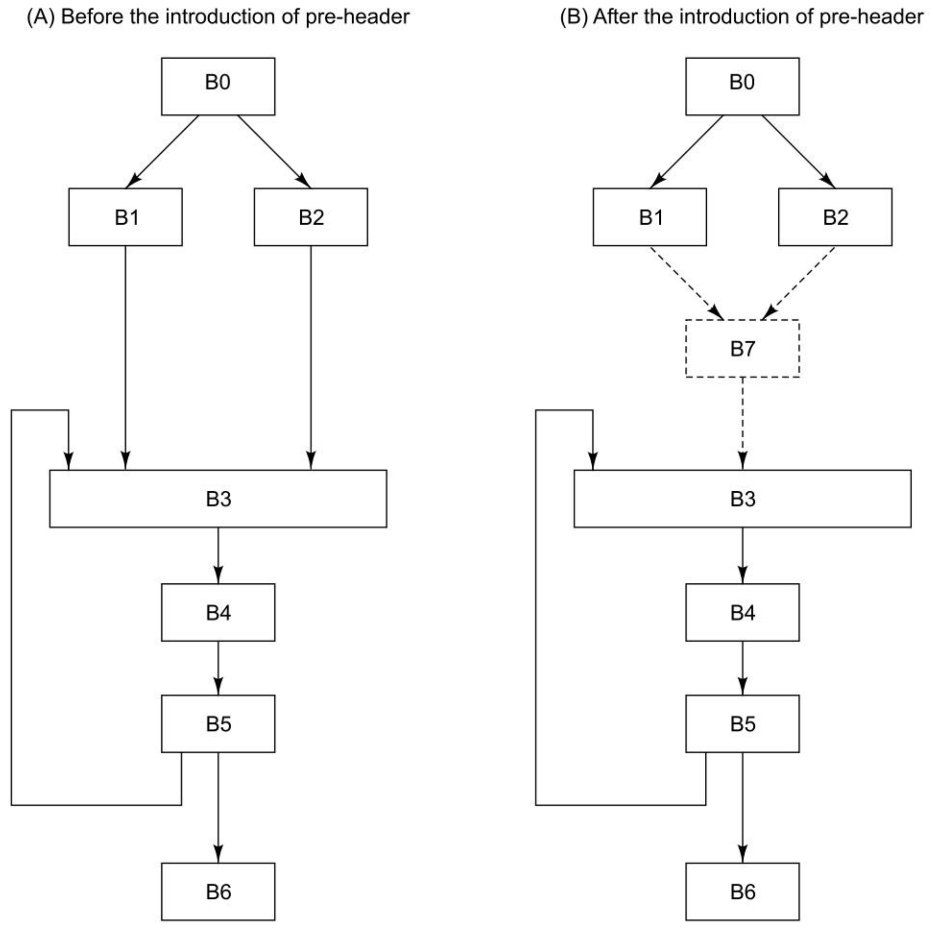

(A) Before the introduction of pre-header
(B) After the introduction of pre-header
B0
B1
B2
B3
B4
B5
B6
B7

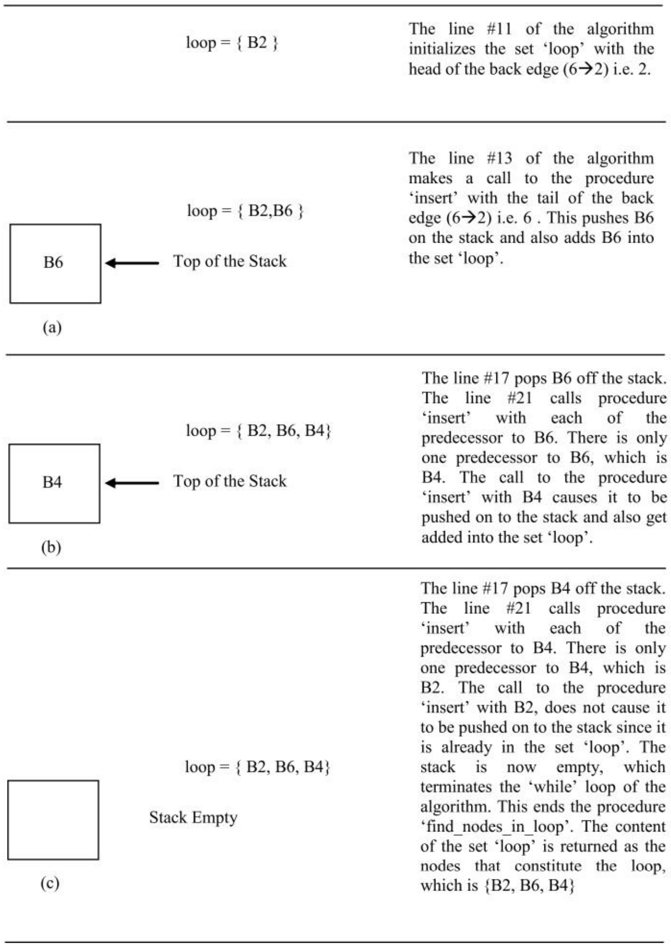
Fig. 7.49 *Pre-header in a loop*

The nodes that constitute a loop in the flow graph are determined by employing Algorithm 7.10. The procedure find_nodes_in_loop($n1$, $n2$) is the one that determines the basic blocks constituting a loop. It is called with the parameters $n1$ and $n2$, where $n1 \rightarrow n2$ is the back edge. A supporting procedure - 'insert', used by 'find_nodes_in_loop' is also shown in Algorithm 7.10.

```
procedure insert(loop,n)
{
   if ( n is not in loop ){
      loop = loop U { n }
      push n on to stack
   }
}

procedure find_nodes_in_loop(n1,n2) /* n1->n2 is the back edge */
{
  loop = { n2 }

  insert( loop, n1 )

15     while ( stack is not empty )
16     {
17        pop node e, the top element of the stack
18
19        for each predecessor of 'e'
20        {
21           insert(loop,e)
22        }
23     }
24     return(loop)
25  }
```

Algorithm 7.10 *Identifying nodes in a loop*

Let's see the working of the algorithm on the flow graph in Fig. 7.48. We take the back edge B6 → B2 into consideration and use Algorithm 7.10 to compute the nodes forming the loop with this back edge. Figure 7.50 shows the sequence of events that occur while using Algorithm 7.10 to compute the constituents of loop for the back edge B6 → B2 of the flow graph in Fig. 7.48.



The line #11 of the algorithm initializes the set 'loop' with the head of the back edge (6→2) i.e. 2.

The line #13 of the algorithm makes a call to the procedure 'insert' with the tail of the back edge (6→2) i.e. 6 . This pushes B6 on the stack and also adds B6 into the set 'loop'.

The line #17 pops B6 off the stack. The line #21 calls procedure 'insert' with each of the predecessor to B6. There is only one predecessor to B6, which is B4. The call to the procedure 'insert' with B4 causes it to be pushed on to the stack and also get added into the set 'loop'.

The line #17 pops B4 off the stack. The line #21 calls procedure 'insert' with each of the predecessor to B4. There is only one predecessor to B4, which is B2. The call to the procedure 'insert' with B2, does not cause it to be pushed on to the stack since it is already in the set 'loop'. The stack is now empty, which terminates the 'while' loop of the algorithm. This ends the procedure 'find_nodes_in_loop'. The content of the set 'loop' is returned as the nodes that constitute the loop, which is {B2, B6, B4}

Fig. 7.50 *Computing the constituents of a loop*

We examined the basic ideas with respect to identifying a loop and its constituents during data flow analysis. This knowledge is useful for any of the loop optimisations. Next, we study about one of the loop optimisations using data flow property called as reaching definitions.

7.2.10.5 Reaching Definitions

Introduction Reaching definitions (RD) is another data flow property that is commonly computed during data flow analysis. In this section, we study about reaching definitions and apply it to perform one of the loop optimisations called as ***loop invariant code motion***.

Consider the input source and the locally optimised TAC shown in Table 7.63 to get an idea of reaching definitions and how that can be used for optimising a loop.

Table 7.63 *Input source and locally optimised code*

Input source

```
int arr[1000];

int func(int a,int b)
{
   int i;
   int n1,n2;

   i=0;

   n1 = a * b ;
   n2 = a - b ;

   while( arr[i] > (n1*n2) )
   {
      i=i+1;
   }


   return(i);
}
```

Locally optimised TAC

```
 (0) proc_begin func
 (1) i := 0
 (2) n1 := a * b
 (3) n2 := a - b

 (4) label .L0
 (5) _t2 := i * 4
 (6) _t3 := &arr
 (7) _t4 := _t3[_t2]
 (8) _t5 := n1 * n2
 (9) if _t4 > _t5 goto .L1

(10) goto .L2

(11) label .L1
(12) i := i + 1
(13) goto .L0

(14) label .L2
(15) return i

(16) goto .L3

(17) label .L3
(18) proc_end func
```

From a visual inspection of the input source and locally optimised TAC in Table 7.63, we can observe the following:

- The variable *n*1 is defined in the quad 2 and used in quad 8. There are no other intervening definitions of *n*1 before its use in quad 8. In other words, the definition of *n*1 that is reaching quad 8 is the one made at line quad 2.
- The variable *n*2 is defined in the quad 3 and used in quad 8. There are no other intervening definitions of *n*2 before its use in quad 8. In other words, the definition of *n*2 that is reaching quad 8 is the one made at line quad 3.
- The quads 4 through 13 comprise the while loop. The definitions of *n*1 and *n*2 that are reaching quad 8 are made outside the loop.

When the 'while' loop spanning from quads 4 through 13 gets executed, the value $n1 * n2$ is computed as many times as the loop executes.

There is an opportunity to optimise the loop by calculating the value $n1 * n2$ before entering the loop and then using it to compare with '*i*' within the loop. This optimisation is possible because the definitions of $n1$ and $n2$ that are reaching quad 8 come from outside the loop (quad 2 and 3 respectively). This is an example of a loop invariant code motion optimisation.

The TAC before and after the loop invariant code motion optimisation is shown in Table 7.64 for the input source in consideration (see Table 7.63).

Table 7.64 *Loop invariant code motion using reaching definitions*

TAC after local optimisation	TAC after loop invariant code motion optimisation
`(0) proc_begin func`	`(0) proc_begin func`
`(1) i := 0`	`(1) i := 0`
`(2) n1 := a * b`	`(2) n1 := a * b`
`(3) n2 := a - b`	`(3) n2 := a - b`
`(4) label .L0`	`(4) _t5 := n1 * n2`
`(5) _t2 := i * 4`	`(5) label .L0`
`(6) _t3 := &arr`	`(6) _t2 := i * 4`
`(7) _t4 := _t3[_t2]`	`(7) _t3 := &arr`
`(8) _t5 := n1 * n2`	`(8) _t4 := _t3[_t2]`
`(9) if _t4 > _t5 goto .L1`	`(9) if _t4 > _t5 goto .L1`
`(10) goto .L2`	`(10) goto .L2`
`(11) label .L1`	`(11) label .L1`
`(12) i := i + 1`	`(12) i := i + 1`
`(13) goto .L0`	`(13) goto .L0`
`(14) label .L2`	`(14) label .L2`
`(15) return i`	`(15) return i`
`(16) goto .L3`	`(16) goto .L3`
`(17) label .L3`	`(17) label .L3`
`(18) proc_end func`	`(18) proc_end func`

The above example showed informally what a reaching definition is and how the loop invariant code motion optimisation could be carried out by knowing the reaching definitions for a particular quad. We conclude this section with a formal statement on a reaching definition.

A definition '*d*' reaches a point '*p*', if there is a path from the point immediately following '*d*' to '*p*' such that '*d*' is not killed along the path.

Suppose, a quad '*q*' uses variables '*v*1' and '*v*2' and the only possible definitions for '*v*1' and '*v*2' come from outside the loop. The quad '*q*' can be considered for performing loop invariant code motion optimisation. Apart from the fact that the reaching definitions are from outside the loop, there are certain other conditions that need to be met in order to make the loop invariant code motion optimisation. We study about those conditions in one of the later sections.

Data Flow Analysis to Compute RD in Intermediate Code In last section, we understood the concept of reaching definitions and how it can be used to perform code motion optimisation. In this section we study about the method that can be used for computing the RD properties of a given program.

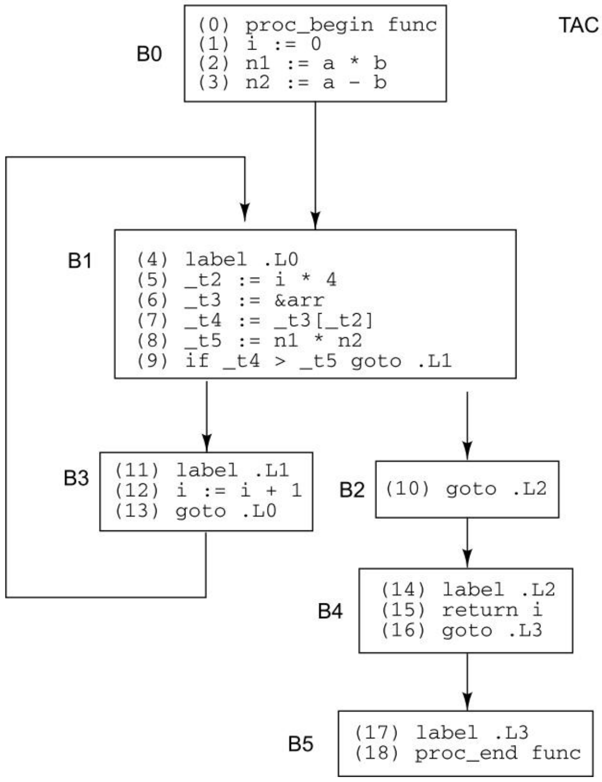
Consider the input source, the corresponding locally optimised TAC and the flow graph shown in Fig. 7.51 to understand some of the terminology that is required for computing the reaching definitions information. The ***universal set of definitions*** L consists of any definition that appears in the statements. In the TAC shown in Fig. 7.51, the definitions can be seen in the quads 1, 2, 3, 5, 6, 7, 8 and 12. Thus, the universal set of definitions for the entire procedure is given by L = {1, 2, 3, 5, 6, 7, 8, 12}.

```
int arr[1000];

int func(int a, int b)
{
  int i, tot;
  int n1, n2;

  i = 0;

  n1 = a * b;
  n2 = a - b;

  while (arr[i] > (n1 * n2))
  {
     i = i + 1;
  }


  return (i);
 }
```

Input source

```
(0)  proc_begin func
(1)  i := 0
(2)  n1 := a * b
(3)  n2 := a - b

(4)  label .L0
(5)  _t2 := i * 4
(6)  _t3 := &arr
(7)  _t4 := _t3[_t2]
(8)  _t5 := n1 * n2
(9)  if _t4 > _t5 goto .L1

(10) goto .L2

(11) label .L1
(12) i := i + 1
(13) goto .L0

(14) label .L2
(15) return i

(16) goto .L3

(17) label .L3
(18) proc_end func
```

TAC after local optimisation



```
B0
(0) proc_begin func
(1) i := 0
(2) n1 := a * b
(3) n2 := a - b

B1
(4) label .L0
(5) _t2 := i * 4
(6) _t3 := &arr
(7) _t4 := _t3[_t2]
(8) _t5 := n1 * n2
(9) if _t4 > _t5 goto .L1

B3
(11) label .L1
(12) i := i + 1
(13) goto .L0

B2
(10) goto .L2

B4
(14) label .L2
(15) return i
(16) goto .L3

B5
(17) label .L3
(18) proc_end func
```

Flow graph

Fig. 7.51 *Input source, TAC after local optimisation and the flow graph*

A block ***generates*** a definition '*d*', if the definition made reaches the end of the block. We use the term rd_GEN[B] to denote the set of definitions generated by a basic block B.

A block ***kills*** all the definitions of a variable '*x*' made outside the block, if it assigns a value to '*x*'. We use the term rd_KILL[B] to denote the definitions killed by a basic block B.

Table 7.65 shows the computation of rd_GEN and rd_KILL sets corresponding to each of the blocks for the TAC shown in Fig. 7.51.

Table 7.65 *rd_GEN/rd_KILL for all the blocks*

#	TAC	rd_GEN	Comments

B0	(0) proc_begin func
	(1) i := 0
	(2) n1 := a * b
	(3) n2 := a - b
		rd_GEN[B0]= { 1,2,3 }
		
		rd_KILL[B0]= { 12 }
			The universal set of definitions that are used in the entire procedure is given by L = { 1,2,3,5,6,7,8,12}
			
			rd_GEN[B0] = { 1,2,3 }, since these definitions reaches the end of the block
			
			rd_KILL[B0] = { 12 } since there is an assignment to 'i' at (12) outside this block.

B1	(4) label .L0
	(5) _t2 := i * 4
	(6) _t3 := &arr
	(7) _t4 := _t3[_t2]
	(8) _t5 := n1 * n2
	(9) if _t4 > _t5 goto .L1
		rd_GEN[B1]= { 5,6,7,8 }
		
		rd_KILL[B1]= { }
			rd_GEN[B1] = { 5,6,7,8 }, since these definitions reache the end of the block
			
			rd_KILL[B1]={ } i.e. empty, since there are no definitions in L outside the block,which define _t2 or _t3 or _t4 or _t5.

B2	(10) goto .L2	rd_GEN[B2] = { }
		
		rd_KILL[B2]= { }
			rd_GEN[B2] = { } i.e. empty
			
			rd_KILL[B2]={ } i.e. empty
			
			Since no definitions are made and as a consequence none are killed either

B3	(11) label .L1
	(12) i := i + 1
	(13) goto .L0
		rd_GEN[B3] = { 12 }
		
		rd_KILL[B3] = { 1 }
			rd_GEN[B3] = { 12 }, since this definition reaches the end of the block
			
			rd_KILL[B3] = { 1 } since there is an assignment to 'i' at (1) outside this block.

B4	(14) label .L2
	(15) return i
	(16) goto .L3
		rd_GEN[B4] = { }
		
		rd_KILL[B4]={ }
			rd_GEN[B4] = { } i.e. empty
			
			rd_KILL[B4]={ } i.e. empty
			
			Since no definitions are made and as a consequence none are killed either

B5	(17) label .L3
	(18) proc_end func
		rd_GEN[B4] = { }
		
		rd_KILL[B4]={ }
			rd_GEN[B4] = { } i.e. empty
			
			rd_KILL[B4]={ } i.e. empty
			
			Since no definitions are made and as a consequence none are killed either

As seen earlier during the discussion on AE, the data flow properties are commonly computed using equations known as ***data flow equations***. By using the data flow equations, one can compute the values of data flow property like RD at the block boundaries. The value of data flow property is then extrapolated within the block by using its value at the block boundary.

The data flow equations for RD express the relationship between the ***rd_IN[B]***, the set of all the definitions reaching the beginning of block 'B' to ***rd_OUT[B]*** the set of definitions reaching the end of block 'B'. The data flow equations for available expressions are as follows:

rd_OUT[B] = rd_GEN[B] ∪ (rd_IN[B] – rd_KILL[B]) rd_IN[B]= ∪ rd_OUT[P] for all the predecessors P of the block	Equation 7.4

Let's use the data flow equations for reaching definitions given by Equation 7.4 and calculate the reaching definitions properties at the start (rd_IN[B]) and end of each block (rd_out[B]) for the sample code shown in Fig. 7.51.

#	TAC	rd_IN	Comments

B0	(0) proc_begin func
	(1) i := 0
	(2) n1 := a * b
	(3) n2 := a - b
		rd_IN[B0]= { }
		
		rd_OUT[B0] = {1,2,3}
			From the data flow equation Equation 7.4
			
			rd_IN[B] = ∪ rd_OUT[P] for all predecessors P
			rd_OUT[B] = rd_GEN[B] ∪ (rd_IN[B] - rd_KILL[B])
			
			There are no predecessors to B0, so rd_IN[B0] = { } i.e. empty
			
			using the values of rd_GEN and rd_KILL for B0 computed earlier
			
			rd_OUT[B0] = {1, 2, 3} ∪ ({ø} - {12})
			rd_OUT[B0]= {1, 2, 3}

B1	(4) label .L0 (5) _t2 := i * 4 (6) _t3 := &arr (7) _t4 := _t3[_t2] (8) _t5 := n1 * n2 (9) if _t4 > _t5 goto .L1		According to Equation 7.4 we have rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block There are 2 predecessors to B1, which are B0 and B3. rd_IN[B1] = rd_OUT[B0] ∪ rd_OUT[B3] Now, rd_OUT[B3] is not yet computed. We are in no position to compute it either, since B1 is a predecessor of B3 and rd_IN[B1] = rd_OUT[B3] We have a cyclic dependency.

We overcome this issue of cyclic dependency issue by using an ***iterative approach*** to solving the data flow equations as we did for the available expressions. In the iterative approach of solving data flow equations for reaching definitions, an initial value for rd_OUT[B] for every block B is assumed. This will help us compute the rd_IN[B], where rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block, even in cases where rd_OUT[P] is not yet computed. The initial value of rd_OUT[B] for every block B that is assumed before the start of all the calculations = rd_GEN[B].

Now, using the initial value of rd_OUT[B], we compute rd_IN[B] for each one of the blocks B, using the equation rd_IN[B] = ∪ rd_OUT[P], for each predecessor P. The initial value of rd_OUT[B] is used in situations like the one given above, where the rd_OUT of the predecessor has not yet been calculated. Once the value of rd_OUT[B] is calculated using Equation 7.4, we would use the calculated value thereafter.

In this manner, we compute rd_IN and rd_OUT for all the blocks. Observe that the rd_IN of some of the blocks like B1, might have been computed directly based on the initial value of rd_OUT for one of the predecessor—B3, while the rd_IN of other blocks like B2, B4, etc. are calculated based on the computed values of rd_OUT of the predecessors. At the end of one round of computation for all the blocks (let's call it iteration 1), we have a set of values of rd_IN/rd_OUT for all the blocks. At this point, note that the rd_OUT for every block contains the computed value and not the initial value that we assigned at the start. Table 7.66 shows rd_IN/rd_OUT calculation for the first iteration corresponding to all the nodes (basic blocks) of the flow graph in Fig. 7.51.

Table 7.66 *rd_IN/rd_OUT for the blocks—iteration #1*

#	TAC	rd_IN	Comments

B0	(0) proc_begin func
	(1) i := 0
	(2) n1 := a * b
	(3) n2 := a - b
		rd_IN[B0]= { }
		
		rd_OUT[B0]= {1, 2, 3}
			From the data flow equation Equation 7.4
			
			rd_IN[B] = ∪ rd_OUT[P] for all predecessors P
			rd_OUT[B] = rd_GEN[B] ∪ (rd_IN[B] - rd_KILL[B])
			
			There are no predecessors to B0, so
			rd_IN[B0] = { } i.e. empty
			
			using the values of rd_GEN and rd_KILL for B0 computed earlier
			
			rd_OUT[B0] = {1, 2, 3} ∪ ({ø} - {12})
			rd_OUT[B0] = {1, 2, 3}

B1	(4) label .L0
	(5) _t2 := i * 4
	(6) _t3 := &arr
	(7) _t4 := _t3[_t2]
	(8) _t5 := n1 * n2
	(9) if _t4 > _t5 goto .L1
		rd_IN[B1]= {1, 2, 3, 12}
		
		rd_OUT[B1] =
		{1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			
			There are 2 predecessors to B1, which are B0 and B3.
			
			rd_IN[B1] = rd_OUT[B0] ∪ rd_OUT[B3]
			
			Initial value of rd_OUT[B3] = rd_GEN[B3] = {12}
			
			rd_IN[B1] = rd_OUT[B0] ∪ rd_OUT[B3]
			
			rd_IN[B1] = {1, 2, 3} ∪ {12}
			rd_IN[B1] = {1, 2, 3, 12}

			rd_OUT[B1] = rd_GEN[B1] ∪ (rd_IN[B1] – rd_KILL[B1])
			rd_OUT[B1] = {5, 6, 7, 8} ∪ ({1, 2, 3, 12} – {})
			rd_OUT[B1] = {1, 2, 3, 5, 6, 7, 8, 12}

B2	(10) goto .L2	rd_IN[B2] = {1, 2, 3, 5, 6, 7, 8, 12}
		rd_OUT[B2] = {1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			There is 1 predecessors to B2, which is B1.
			rd_IN[B2] = rd_OUT[B1]
			rd_IN[B2] = {1, 2, 3, 5, 6, 7, 8, 12}
			rd_OUT[B2] = rd_GEN[B2] ∪ (rd_IN[B2] – rd_KILL[B2])
			rd_OUT[B2] = { } ∪ ({1, 2, 3, 5, 6, 7, 8, 12} – { })
			rd_OUT[B2] = {1, 2, 3, 5, 6, 7, 8, 12}

B3	(11) label .L1
	(12) i := i + 1
	(13) goto .L0
		rd_IN[B3] = {1, 2, 3, 5, 6, 7, 8, 12}
		rd_OUT[B3] = {2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			There is 1 predecessors to B3, which is B1.
			rd_IN[B3] = rd_OUT[B1]
			rd_IN[B3] = {1, 2, 3, 5, 6, 7, 8, 12}
			rd_OUT[B3] = rd_GEN[B3] ∪ (rd_IN[B3] – rd_KILL[B3])
			rd_OUT[B3] = {12} U ({5, 6, 7, 8, 1, 2, 3, 12} – {1})
			rd_OUT[B3] = {2, 3, 5, 6, 7, 8, 12}

B4	(14) label .L2
	(15) return i
	(16) goto .L3
		rd_IN[B4] = {1, 2, 3, 5, 6, 7, 8, 12}
		rd_OUT[B4] = {1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			There is 1 predecessors to B4, which is B2.
			rd_IN[B4] = rd_OUT[B2]
			rd_IN[B4] = {1, 2, 3, 5, 6, 7, 8, 12}

			rd_OUT[B4] = rd_GEN[B4] U (rd_IN[B4] - rd_KILL[B4])
			
			rd_OUT[B4] = { } ∪ ({1, 2, 3, 5, 6, 7, 8, 12} - { })
			
			rd_OUT[B4] = {1, 2, 3, 5, 6, 7, 8, 12}

B5	(17) label .L3
	(18) proc_end func
		rd_IN[B5] = {1, 2, 3, 5, 6, 7, 8, 12}
		
		rd_OUT[B5] = {1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			
			There is 1 predecessors to B5, which is B4.
			
			rd_IN[B5] = rd_OUT[B4]
			rd_IN[B5] = {1, 2, 3, 5, 6, 7, 8, 12}
			
			rd_OUT[B5] = rd_GEN[B5] U (rd_IN[B5] - rd_KILL[B5])
			
			rd_OUT[B5] = { } ∪ ({1, 2, 3, 5, 6, 7, 8, 12} - { })
			
			rd_OUT[B5] = {1, 2, 3, 5, 6, 7, 8, 12}

Next, we do the computation of rd_IN and rd_OUT for all the blocks again (iteration 2) using the same equations defined in Equation 7.4. In this second iteration, the value of rd_IN for the blocks like B1 change, since rd_OUT[B3] would now be the value computed in the first iteration, i.e. {2, 3, 5, 6, 7, 8, 12} as opposed to initial value {12} used in the first iteration. The changes in rd_IN[B1] might have a ripple effect changing rd_OUT[B1], rd_IN[B3] and then rd_OUT[B3]. Thus the second iteration yields a set of values of rd_IN/rd_OUT for all the blocks.

Table 7.67 *rd_IN/rd_OUT for the blocks—iteration #2*

#	TAC	rd_IN	Comments

B0	(0) proc_begin func
	(1) i := 0
	(2) n1 := a * b
	(3) n2 := a - b
		rd_IN[B0] = { }
		
		rd_OUT[B0] = {1, 2, 3}
			From the data flow equation Equation 7.4
			
			rd_IN[B] = U rd_OUT[P] for all predecessors P
			rd_OUT[B] = rd_GEN[B] ∪ (rd_IN[B] - rd_KILL[B])
			
			There are no predecessors to B0, so rd_IN[B0] = { } i.e. empty
			
			using the values of rd_GEN and rd_KILL for B0 computed earlier
			
			rd_OUT[B0] = {1, 2, 3} ∪ ({ø} - {12})
			
			rd_OUT[B0] = {1, 2, 3}

B1	(4) label .L0
	(5) _t2 := i * 4
	(6) _t3 := &arr
	(7) _t4 := _t3[_t2]
	(8) _t5 := n1 * n2
	(9) if _t4 > _t5
	goto .L1
		rd_IN[B1] =
		{1, 2, 3, 5, 6, 7, 8, 12}
		
		rd_OUT[B1] =
		{1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have
			
			rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			
			There are 2 predecessors to B1, which are B0 and B3.
			
			rd_IN[B1] = rd_OUT[B0] ∪ rd_OUT[B3]
			
			The value of rd_OUT[B3] computed in the previous iteration= {2, 3, 5, 6, 7, 8, 12}
			
			rd_IN[B1] = rd_OUT[B0] ∪ rd_OUT[B3]
			
			rd_IN[B1] = {1, 2, 3} ∪ {{2, 3, 5, 6, 7, 8, 12}
			
			rd_IN[B1] = {1, 2, 3, 5, 6, 7, 8, 12}
			
			rd_OUT[B1] = rd_GEN[B1] ∪ (rd_IN[B1] – rd_KILL[B1])
			
			rd_OUT[B1] = {5, 6, 7, 8} ∪ ({1, 2, 3, 5, 6, 7, 8, 12} – { })
			
			rd_OUT[B1] = {1, 2, 3, 5, 6, 7, 8, 12}

B2	(10) goto .L2	rd_IN[B2] =
		{1, 2, 3, 5, 6, 7, 8, 12}
		
		rd_OUT[B2] =
		{1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have
			
			rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			
			There is 1 predecessor to B2, which is B1.
			
			rd_IN[B2] = rd_OUT[B1]
			rd_IN[B2] = {1, 2, 3, 5, 6, 7, 8, 12}
			
			rd_OUT[B2] = rd_GEN[B2] ∪ (rd_IN[B2] – rd_KILL[B2])
			
			rd_OUT[B2] = { } ∪ ({1, 2, 3, 5, 6, 7, 8, 12} – { })
			
			rd_OUT[B2] = { 1,2,3,5,6,7,8,12 }

B3	(11) label .L1
	(12) i := i + 1
	(13) goto .L0
		rd_IN[B3] =
		{1, 2, 3, 5, 6, 7, 8, 12}
		
		rd_OUT[B3] =
		{2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have
			
			rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			
			There is 1 predecessors to B3, which is B1.
			
			rd_IN[B3] = rd_OUT[B1]
			rd_IN[B3] = {1, 2, 3, 5, 6, 7, 8, 12}
			
			rd_OUT[B3] = rd_GEN[B3] ∪ (rd_IN[B3] – rd_KILL[B3])
			
			rd_OUT[B3] = {12} ∪ ({5, 6, 7, 8, 1, 2, 3, 12} – {1})
			
			rd_OUT[B3] = {2, 3, 5, 6, 7, 8, 12}

B4	(14) label .L2
	(15) return i
	(16) goto .L3
		rd_IN[B4] =
		{1, 2, 3, 5, 6, 7, 8, 12}
		
		rd_OUT[B4] =
		{1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have
			
			rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block
			
			There is 1 predecessors to B4, which is B2.
			
			rd_IN[B4] = rd_OUT[B2]
			rd_IN[B4] = {1, 2, 3, 5, 6, 7, 8, 12}
			
			rd_OUT[B4] = rd_GEN[B4] ∪ (rd_IN[B4] – rd_KILL[B4])
			
			rd_OUT[B4] = { } ∪ ({1, 2, 3, 5, 6, 7, 8, 12} – { })
			
			rd_OUT[B4] = {1, 2, 3, 5, 6, 7, 8, 12}

B5	(17) label .L3
	(18) proc_end func
		rd_IN[B5] =
		{1, 2, 3, 5, 6, 7, 8, 12}
		
		rd_OUT[B5] =
		{1, 2, 3, 5, 6, 7, 8, 12}
			According to Equation 7.4 we have
			
			rd_IN[B] = U rd_OUT[P] for all the predecessors P of the block
			
			There is 1 predecessor to B5, which is B4.
			
			rd_IN[B5] = rd_OUT[B4]
			rd_IN[B5] = {1, 2, 3, 5, 6, 7, 8, 12}
			
			rd_OUT[B5] = rd_GEN[B5] ∪ (rd_IN[B5] – rd_KILL[B5])
			
			rd_OUT[B5] = { } U ({1, 2, 3, 5, 6, 7, 8, 12} – { })
			
			rd_OUT[B5] = {1, 2, 3, 5, 6, 7, 8, 12}

Table 7.68 shows the values of rd_OUT as calculated in the first and second iteration corresponding to all the basic blocks of the flow graph. Observe that the values of rd_OUT for none of the blocks have changed from the first to second iteration. This signals us to half the iterations calculating rd_IN/e_OUT for the blocks, since rd_OUT of all the blocks have reached a steady state.

Table 7.68 *rd_OUT in the first and second iteration*

Block #	rd_OUT in iteration #1	rd_OUT in iteration #2
0	rd_OUT[B0] = {1, 2, 3}	rd_OUT[B0] = {1, 2, 3}
1	rd_OUT[B1] = {1, 2, 3, 5, 6, 7, 8, 12}	rd_OUT[B1] = {1, 2, 3, 5, 6, 7, 8, 12}
2	rd_OUT[B2] = {1, 2, 3, 5, 6, 7, 8, 12}	rd_OUT[B2] = {1, 2, 3, 5, 6, 7, 8, 12}
3	rd_OUT[B3] = {2, 3, 5, 6, 7, 8, 12}	rd_OUT[B3] = {2, 3, 5, 6, 7, 8, 12}
4	rd_OUT[B4] = {1, 2, 3, 5, 6, 7, 8, 12}	rd_OUT[B4] = {1, 2, 3, 5, 6, 7, 8, 12}
5	rd_OUT[B5] = {1, 2, 3, 5, 6, 7, 8, 12}	rd_OUT[B5] = {1, 2, 3, 5, 6, 7, 8, 12}

Figure 7.52 shows the flow graph annotated with the values of rd_IN, rd_GEN, rd_KILL and rd_OUT for each of the blocks—B0 through B5.

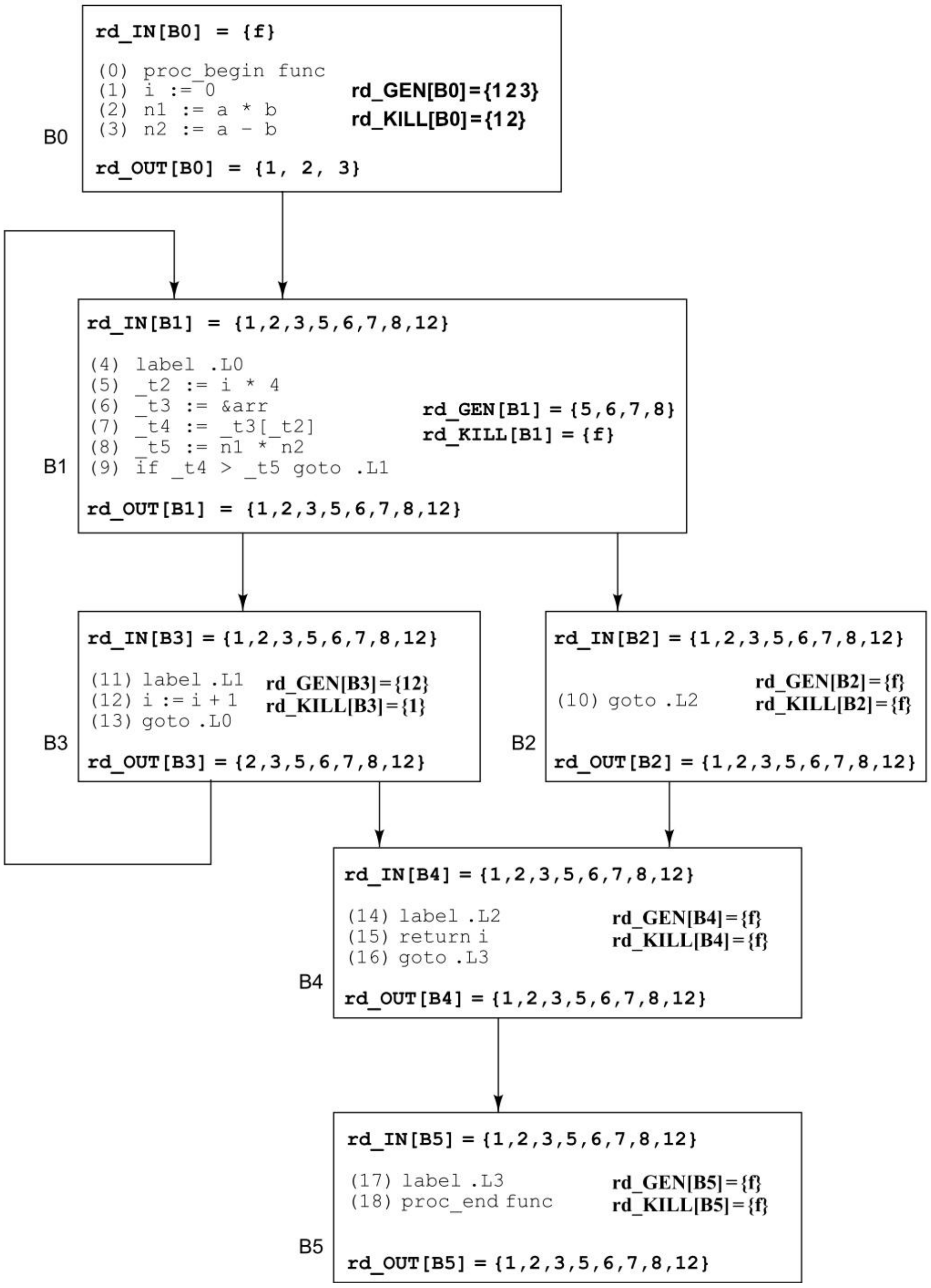


Fig. 7.52 *Flow graph annotated with reaching definition information*

Algorithm 7.11 summarises the computation of reaching definitions (rd_IN/rd_OUT) using the iterative approach of solving data flow equations that we discussed above.

```
/* Initialize rd_OUT for all blocks */
for every block B {
        rd_OUT[B]= rd_GEN[B]
}

steady_state = FALSE

while (steady_state = = FALSE) {
        steady_state = TRUE
        for every block B {

             /* rd_IN */
             rd_IN[B] = ∪ rd_OUT[P] for all the predecessors P of the block

             /* saving rd_OUT to later check if we have reached steady state */
             saved_rd_OUT = rd_OUT

             /* computing rd_OUT */
             rd_OUT[B] = rd_GEN[B] ∪ ( rd_IN[B] - rd_KILL[B] )

             /* Checking for a steady state of rd_OUT */
             if(saved_rd_OUT[B] ! = rd_OUT[B]){
                  steady_state = FALSE
             }
        }
}
```

Algorithm 7.11 *Reaching definitions computation using the iterative approach*

Algorithm 7.11 works for the input sources involving loops and also the ones without the loops. Observe in the algorithm that the final values of rd_IN and rd_OUT for all the blocks in the procedure are arrived at in an iterative fashion. For an input source without any loops, the final values of rd_IN and rd_OUT for each block can be arrived at in the first iteration, if the computation is made in the order of flow of control, where we compute rd_IN / rd_OUT for a block B only after rd_IN/rd_OUT for all its predecessors have been computed. In such cases, the second iteration is performed only to confirm that the values of rd_OUT have reached a steady state.

The reaching definition information in the form of rd_IN set at the block level is extrapolated to the quad level and stored in a data structure called ***use-definition (ud) chain***.

The ud-chain is a set holding all definitions reaching a quad, for each variable used in the quad. Let's pick a quad and see its ud-chain, to understand it. Let's take quad (8) $t5 := n1 * n2$ in the TAC after local optimisation in the flow graph, Fig. 7.52 . The ud-chains for the same are shown below.

```
ud_chain (8, n1) = {2}
ud_chain (8, n2) = {3}
```

This tells us that the definition of '*n*1' reaching the quad (8) comes from the quad (2). Similarly, the definition of '*n*2' reaching quad (8) comes from the quad (3).

Let's take another quad to see a case where there are more than one definitions reaching the quad—the quad (5) $i := i + 1$ in Fig. 7.52. The quad (5) uses one variable '*i*', the other argument used is a constant '1'. The ud-chain for the quad (5) is shown below.

```
ud_chain (5, i) = {1, 12}
```

The ud-chain above tells us that the definition of '*i*' reaching the quad (5) comes from either the quad (1) or quad (12). The quad (1) provides the initial value when the loop is entered for the first time and quad (12) provides the value thereafter.

The ud-chain at a quad '*q*' for the usage of a variable '*v*' can be directly calculated from rd_IN of the block, by selecting the sub-set of definitions corresponding to the variable '*v*' within rd_IN. In cases where there is a definition of the variable '*v*' within the current block before the quad '*q*', at say, *q*0, then the ud-chain (*q*, *v*) = *q*0 (i.e.) *q*0 is the only definition that reaches *q* and the elements in rd_IN for '*v*' are ignored. In scenarios containing multiple definitions of variable '*v*' in the current block, then '*q*0' corresponds to the last definition of the variable '*v*'.

Consider the above example for which we have already constructed the reaching definitions information. Table 7.69 shows the TAC annotated with the rd_IN information that we computed above.

Table 7.69 *TAC with RD information*

```
/* rd_IN[B0] = {ø} */
(0) proc_begin func
(1) i := 0
(2) n1 := a * b
(3) n2 := a - b

/* rd_IN[B1] = {1, 2, 3, 5, 6, 7, 8, 12} */
(4) label .L0
(5) _t2 := i * 4
(6) _t3 := &arr
(7) _t4 := _t3[_t2]
(8) _t5 := n1 * n2
(9) if _t4 > _t5 goto .L1

/* rd_IN[B2] = {1, 2, 3, 5, 6, 7, 8, 12} */
(10) goto .L2

/* rd_IN[B3] = {1, 2, 3, 5, 6, 7, 8, 12} */
(11) label .L1
(12) i := i + 1
(13) goto .L0

/* rd_IN[B4] = {1, 2, 3, 5, 6, 7, 8, 12} */
(14) label .L2
(15) return i
(16) goto .L3

/* rd_IN[B5] = {1, 2, 3, 5, 6, 7, 8, 12} */
(17) label .L3
(18) proc_end func
```

The ud-chains for the quads of blocks B1 and B3 in the TAC shown in Table 7.69 are summarised below in Table 7.70. The ud-chains are not relevant for the quads using certain operators like label, proc_begin, proc_end and hence are not listed in Table 7.70.

Table 7.70 *ud-chain information*

Block	Quad	ud-chain Information	Explanation
B1	`(5) _t2 := i * 4`	`ud_chain (5, i) = {1, 12}`	The definitions corresponding to 'i' in the rd_IN[B1] are {1, 12}. There are no definitions of 'i' preceding the quad #5 within the block B1. Hence the definitions of 'i' that are reaching the quad 5 are the ones made in {1, 12}.
B1	`(6) _t3 := &arr`	`None`	The ADDR_OF operator fetches the memory address of the operand 'arr'. It is an l-value and cannot be defined by any previous quad. It is treated like a constant assignment

B1	`(7) _t4 := _t3[_t2]`	`ud_chain (7,_t3) = {6}`
		`ud_chain (7,_t2) = {5}`
			There is a definition of '_t3' in quad #6 preceding the quad #7 within the block B1. Hence, the definition of '_t3' that is reaching the quad 7 is the one made in {6}.
			
			There is a definition of '_t2' in quad #5 preceding the quad #7 within the block B1. Hence, the definitions of '_t2' that is reaching the quad 7 is the one made in {5}.

B1	`(8) _t5 := n1 * n2`	`ud_chain (8, n1) = {2}`
		`ud_chain (8, n2) = {3}`
			The definitions corresponding to 'n1' in the rd_IN[B1] is {2}. There are no definitions of 'n1' preceding the quad #8 within the block B1. Hence, the definition of 'n1' that is reaching the quad 8 is the one made in {2}.
			
			The definitions corresponding to 'n2' in the rd_IN[B1] is {3}. There are no definitions of 'n2' preceding the quad #8 within the block B1. Hence, the definition of 'n2' that is reaching the quad 8 is the one made in {3}.

B1	`(9) if _t4 > _t5 goto .L1`	`ud_chain (9, _t4) = {7}`
		`ud_chain (9, _t5) = {8}`
			The definitions corresponding to '_t4' in the rd_IN[B1] are None. There is a definition of '_t4' in quad #7 preceding the quad #9 within the block B1. Hence, the definition of '_t4' that is reaching the quad 9 is the one made in {7}.
			
			The definitions corresponding to '_t5' in the rd_IN[B1] are None. There is a definition of '_t5' in quad #8 preceding the quad #9 within the block B1. Hence, the definition of '_t5' that is reaching the quad 9 is the one made in {8}.

B3	(12) i := i + 1	ud_chain (12, i) = {1, 12}	The definitions corresponding to 'i' in the rd_IN[B3] are {1, 12}. There are no definitions of 'i' preceding the quad #12 within the block B3. Hence, the definitions of 'i' that are reaching the quad 12 are the ones made in {1, 12}. The definitions of 'i' that are reaching the quad 15 is the one made in {1, 12}.

The ud-chain is used to perform optimisations like the loop invariant code motion optimisation.

Loop Invariant Code Motion Optimisation using RD Analysis In this section, we discuss about using the ud-chain obtained from RD analysis for performing the code motion optimisation in the loops. As mentioned earlier, the code motion optimisation deals with moving the loop invariant statements out of the loop. Moving the loop invariant statements out of the loop reduces the amount of computation done in an iteration of the loop. This can potentially improve the performance of the program manifold.

There are two steps required for performing the loop invariant code motion optimisation. They are:

1. The detection of loop invariant statements in the loop. This is based on the ud-chain information obtained from the reaching definition analysis discussed previously.
2. The moving of the loop invariant statements to the pre-header of the loop. The loop invariant statements are moved to the pre-header on ascertaining certain conditions.

We study about each of these steps in detail in the following paragraphs.

Step 1: Detection of loop invariant statements

A loop invariant statement computes a value that does not change throughout the execution of the loop. In a more formal way, a statement 's: $x := y + z$' in a loop L is considered as loop invariant if one of the following conditions hold good:

1. All the reaching definitions of 'y' and 'z' at 's' are from outside the loop as indicated by the ud chains for the quad q.
2. The operands 'y' and 'z' are constants.

The identification of loop invariant statements could take multiple passes as the following suggests. Consider a statement $s : x := y + z$ within a loop L. Suppose the definitions of 'y' and 'z' that are reaching 's' are all from outside the loop. The computation ($y + z$) will yield a value 'x', which remains constant throughout the execution of the loop. The statement 's' is a Loop invariant. Suppose there is another statement $s1 : m := x + k$ right after the statement 's'. Let's assume that reaching definitions of 'k' are outside the loop. The statement '$s1$' is also a loop invariant since the computation '$x + k$' remains constant throughout the execution of the loop. The statement 's' in the above example can be detected in the first pass of the quads in the loop. The statement '$s1$' can be detected in the second pass of the quads, after 's' has been moved out of the loop.

Consider the flow graph in Fig. 7.52 and the corresponding ud-chain information computed earlier in Table 7.70. Table 7.71 shows a section of the ud-chain information corresponding to the statements of the basic blocks B1 and B3 that form a loop. From Table 7.71, the statement (8) can be concluded as a loop invariant, since the reaching definitions for $n1$ and $n2$ are from statements (2) and (3), which are outside the loop. The statement (6) is also a loop invariant, since the operator '&' on any variable yields a constant.

Table 7.71 *ud-chain information for the statements in B1 and B3*

Block #	TAC statements	ud-Chain information	Comments
B1	`(5) _t2 := i * 4`	`ud_chain (5, i) = {1, 12}`	
	`(6) _t3 := &arr`	`None`	`It is a constant assignment. This statement is a loop invariant`
	`(7) _t4 := _t3[_t2]`	`ud_chain (7, _t3) = {6}` `ud_chain (7, _t2) = {5}`	
	`(8) _t5 := n1 * n2`	`ud_chain (8, n1) = {2}` `ud_chain (8, n2) = {3}`	`This statement is a loop invariant`
	`(9) if _t4 > _t5 goto .L1`	`ud_chain (9, _t4) = {7}` `ud_chain (9, _t5) = {8}`	
B3	`(11) label .L1`		
	`(12) i := i + 1`	`ud_chain (12, i) = {1, 12}`	
	`(13) goto .L0`		

The loop invariant statements are thus identified using the ud-chain information as shown in the above example.

Step 2: Moving the Loop Invariant Statements to the Pre-header

All the loop invariant statements identified in step(1) cannot be moved unconditionally into pre-header of the loop. The loop invariant statements need to meet several conditions in order to qualify for movement to the pre-header of the loop. These conditions stem from the idea that optimisation cannot result in incorrect code under any circumstances.

For a statement 's: $a = b + c$', to be moved into the pre-header, the following are the conditions that should be met.

1. There should be no other statement '$s1$', which defines 'a' within L.
2. The reaching definition for all the uses of 'a' in the loop should be from 's' only.
3. The statement 's' should be in a block that dominates all the exits of the loop L.

The loop invariant statements identified in step(1), meeting the above criteria are moved into pre-header of the loop for accomplishing loop invariant code motion optimisation.

The following examples illustrate how violation of any one of the conditions could potentially lead to erroneous code generation.

Table 7.72 shows an input source that has a loop in which the variable 'i' varies. The copy statement 'flag = 0' at line #12 is a loop invariant statement. However, moving that to the pre-header of the loop causes the function 'func' to behave incorrectly. The function might return '0', irrespective of whether the condition' (arr[i] < a)' at line #14 is true or not for the loop exiting value of 'i'. This loop invariant quad at #5 flag := 0, cannot be moved into a pre-header block since there is another statement which defines 'flag' within the loop. It violates the condition 1, i.e. there should be no other statement '$s1$', which defines 'flag' within the loop L.

Table 7.72 *Condition 1 violated*

Input source	TAC after local optimisation	Incorrect movement of loop invariant statement to the pre-header of the loop
`1 int arr [100];`	`(0) proc_begin func`	`(0) proc_begin func`
`2`	`(1) i := 0`	`(1) i := 0`
`3`	`(2) label .L0`	`(1a) flag := 0`
`4 int func (int a, int b)`	`(3) goto .L1`	`(2) label .L0`
`5 {`	`(4) goto .L6`	`(3) goto .L1`
`6     int i, flag;`	`(5) label .L1`	`(4) goto .L6`
`7`	`(6) flag := 0`	`(5) label .L1`
`8     i = 0;`	`(7) _t0 := i * 4`	`(6)`
`9`	`(8) _t1 := &arr`	`(7) _t0 := i * 4`
`10    while (1 == 1){`	`(9) _t2 := _t1[_t0]`	`(8) _t1 := &arr`
`11`	`(10) if _t2 < a goto .L2`	`(9) _t2 := _t1[_t0]`
`12        flag=0;`	`(11) goto .L3`	`(10) if _t2 < a goto .L2`
`13`	`(12) label .L2`	`(11) goto .L3`
`14        if (arr[i] < a){`	`(13) flag := 1`	`(12) label .L2`
`15            flag=1;`	`(14) label .L3`	`(13) flag := 1`
`16        }`	`(15) _t3 := i * 4`	`(14) label .L3`
`17`	`(16) _t4 := &arr`	`(15) _t3 := i * 4`
`18`	`(17) _t5 := _t4[_t3]`	`(16) _t4 := &arr`
`19        if ( arr[i] >= b){`	`(18) if _t5 >= b goto .L4`	`(17) _t5 := _t4[_t3]`
`20            break;`	`(19) goto .L5`	`(18) if _t5 >= b goto .L4`
`21        }`	`(20) label .L4`	`(19) goto .L5`
`22`	`(21) goto .L6`	`(20) label .L4`
`23        i = i+1;`	`(22) label .L5`	`(21) goto .L6`
`24`	`(23) i := i + 1`	`(22) label .L5`
`25    }`	`(24) goto .L0`	`(23) i := i + 1`
`26    return (flag);`	`(25) label .L6`	`(24) goto .L0`
`27 }`	`(26) return flag`	`(25) label .L6`
	`(27) goto .L7`	`(26) return flag`
	`(28) label .L7`	`(27) goto .L7`
	`(29) proc_end func`	`(28) label .L7`
		`(29) proc_end func`

Table 7.73 shows an input source that has a loop in which the variable ‘change’ could have a value of 1 or 0 in the first iteration, and 0 in all other iterations. The function f1 is called with ‘change’ as a parameter. The copy statement ‘change = 0’ at line#19 is a loop invariant statement. However moving that to the pre-header of the loop causes the invocation of function ‘f1’ with change taking the value of 0 as a parameter for all the iterations including the first. This causes the incorrect behaviour of the program in the event of the condition ($a > b$) being true. The correct behaviour of the program would be to invoke function ‘f1’ with ‘change’ taking the value of 1 as a parameter for the first iteration, in the event of the condition (a > b) being true.

This loop invariant quad at #15 change := 0, cannot be moved into a pre-header block since it violates the condition 2, i.e. the reaching definition for all the uses of ‘change’ in the loop should be from quad #15 only. The reaching definitions at quad #11 (param change), which specifies the parameter for the invocation of f1, are from #1, #5 and #15. One of the prerequisites for moving the loop invariant quad #15 to the pre-header of the loop in the form of condition 2 is that the reaching definition for all the uses of ‘change’ in the loop (quad #11) should be from the quad #15 only. Clearly, the condition 2 has been violated in this case and hence the quad #15 cannot be moved to the pre-header of the loop.

Table 7.73 *Condition 2 violated*

Input source	TAC after local optimisation	Incorrect movement of loop invariant statement to the pre-header of the loop
1 `int arr[100];`	`(0) proc_begin func`	`(0) proc_begin func`
2	`(1) change := 0`	`(1) change := 0`
3 `int f1( );`	`(2) if a > b goto .L0`	`(2) if a > b goto .L0`
4	`(3) goto .L1`	`(3) goto .L1`
5 `int func (int a, int b)`	`(4) label .L0`	`(4) label .L0`
6 `{`	`(5) change := 1`	`(5) change := 1`
7 `    int change, ret;`	`(6) label .L1`	`(6) label .L1`
8	`(7) label .L2`	`(7) label .L2`
9 `    change=0;`	`(8) goto .L3`	`(7a) change := 0`
10 `    if (a > b){`	`(9) goto .L6`	`(8) goto .L3`
11 `        change=1;`	`(10) label .L3`	`(9) goto .L6`
12 `    }`	`(11) param change`	`(10) label .L3`
13	`(12) call f1 4`	`(11) param change`
14	`(13) ret := _t0`	`(12) call f1 4`
15 `    while (1 == 1){`	`(14) retrieve _t0`	`(13) ret := _t0`
16	`(15) change := 0`	`(14) retrieve _t0`
17 `        ret = f1(change);`	`(16) if _t0 == 0 goto .L4`	`(15)`
18	`(17) goto .L5`	`(16) if _t0 == 0 goto .L4`
19 `        change = 0;`	`(18) label .L4`	`(17) goto .L5`
20	`(19) goto .L6`	`(18) label .L4`
21 `        if (ret == 0){`	`(20) label .L5`	`(19) goto .L6`
22 `            break;`	`(21) goto .L2`	`(20) label .L5`
23 `        }`	`(22) label .L6`	`(21) goto .L2`
24	`(23) goto .L7`	`(22) label .L6`
25 `    }`	`(24) label .L7`	`(23) goto .L7`
26 `    return;`	`(25) proc_end func`	`(24) label .L7`
27 `}`		`(25) proc_end func`

Table 7.74 shows an input source that has a loop in which the variable '*i*' varies. The copy statement 'flag = 1' at line #17 (quad 16) is a loop invariant statement. However, moving that to the pre-header of the loop causes the function 'cap_it' to always return the value 1, irrespective of whether the condition '(arr[*i*] > *a*)' at line #11 is true or not.

Table 7.74 *Condition 3 violated*

Input source	TAC after local optimisation	Incorrect movement of loop invariant statement to the pre-header of the loop
`int arr[100];`	`(0) proc_begin cap_it`	`(0) proc_begin cap_it`
	`(1) i := 0`	`(1) i := 0`
	`(2) flag := 0`	`(2) flag := 0`
`int cap_it(int a)`		`(2a) flag := 1`
`{`		
`    int i, flag;`	`(3) label .L0`	`(3) label .L0`
	`(4) if i < 100 goto .L1`	`(4) if i < 100 goto .L1`
`    i = 0;`		
`    flag = 0;`	`(5) goto .L4`	`(5) goto .L4`
`    while (i < 100){`	`(6) label .L1`	`(6) label .L1`
	`(7) _t0 := i * 4`	`(7) _t0 := i * 4`
`        if (arr[i] > a){`	`(8) _t1 := &arr`	`(8) _t1 := &arr`
`            arr[i] = a;`	`(9) _t2 := _t1[_t0]`	`(9) _t2 := _t1[_t0]`
	`(10) if _t2 > a goto .L2`	`(10) if _t2 > a goto .L2`
`            /* Loop Invariant */`		
`            flag=1;`	`(11) goto .L3`	`(11) goto .L3`
`        }`		
	`(12) label .L2`	`(12) label .L2`
`        i = i + 1;`	`(13) _t3 := i * 4`	`(13) _t3 := i * 4`
`    }`	`(14) _t4 := &arr`	`(14) _t4 := &arr`
	`(15) _t4[_t3] := a`	`(15) _t4[_t3] := a`
`    return (flag);`	`(16) flag := 1`	`(16)`
`}`	`(17) label .L3`	`(17) label .L3`
	`(18) i := i + 1`	`(18) i := i + 1`
	`(19) goto .L0`	`(19) goto .L0`
	`(20) label .L4`	`(20) label .L4`
	`(21) return flag`	`(21) return flag`
	`(22) goto .L5`	`(22) goto .L5`
	`(23) label .L5`	`(23) label .L5`
	`(24) proc_end cap_it`	`(24) proc_end cap_it`

The flow graph of the input source is shown in Fig. 7.53. We can gather from the flow graph that the block B5 in which the loop invariant quad '(16) flag := 1' is found, does not dominate the loop exit block B1. The loop invariant statement #16 cannot be moved into a pre-header block since it violates the condition 3, i.e. the block in which it exists does not dominate all the exits of the loop.

The reader is advised to check if all the three conditions are satisfied in the loop invariant code optimization example shown in Table 7.64.

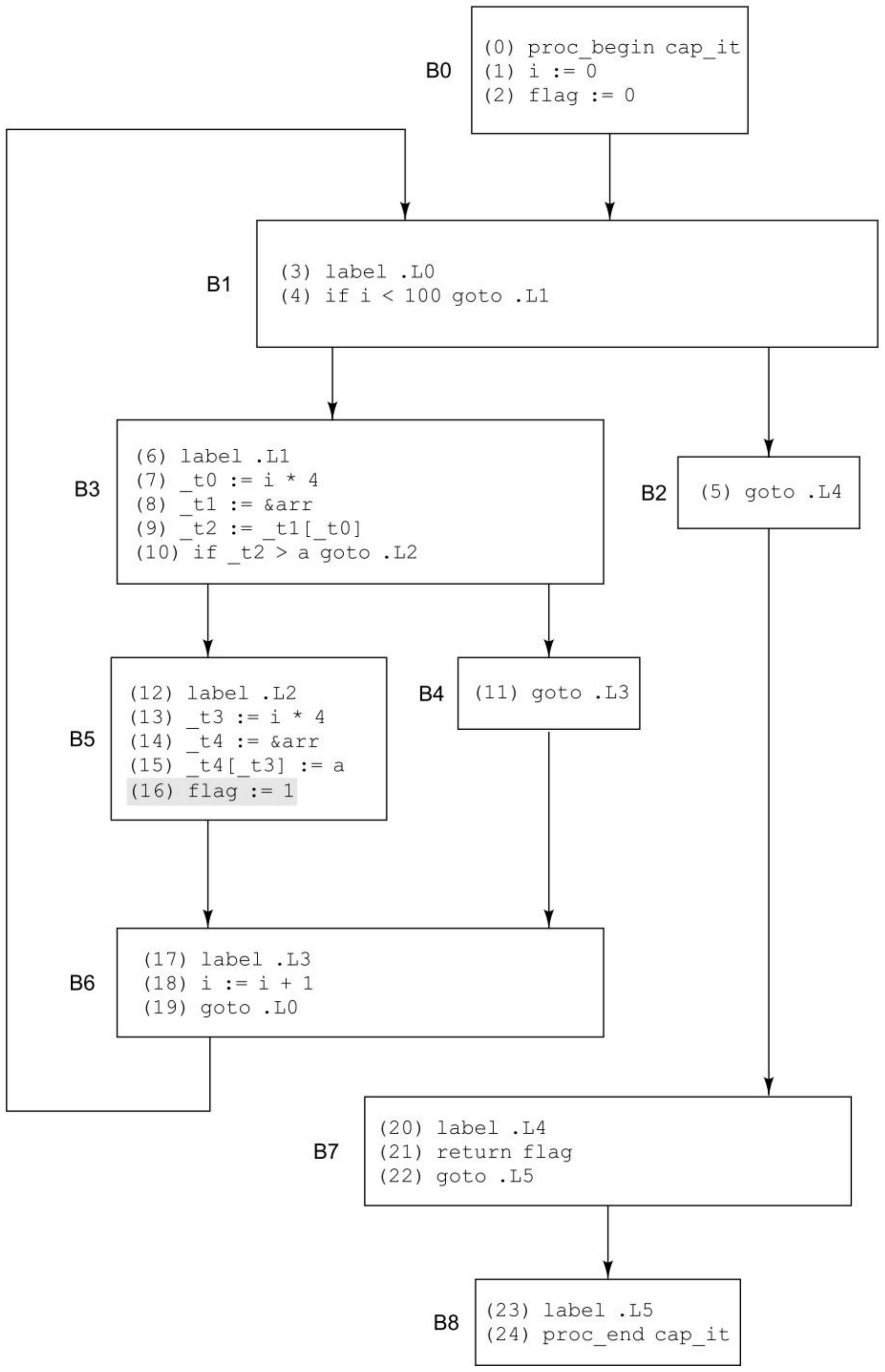


Fig. 7.53 *Flow graph*

Example 6—Global Code Motion Using Reaching Definition Analysis This section demonstrates the toy C compiler (mycc) performing global loop invariant code motion using the reaching definition information. The reaching definition information was gathered by using the algorithms explained in the preceding section. The RD information in the form of ud-chains is put to use to move the loop invariant code to the pre-header on ascertaining that the three conditions mentioned earlier have been fulfilled.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC (b) the locally optimised TAC and (c) the TAC after global loop invariant code motion. The dialog below shows ‘mycc’ taking in some sample input C sources having loops and printing out the above information as the output.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$  g++   -DICGEN  -g  -Wall  ic_gen.cc  optimize.cc  target_code_gen.cc  mycc.cc  semantic_
analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Sample Input C file with Loop invariants that can be moved out
$ cat -n test.cm.1.c
     1  int arr[1000];
     2
     3  int func (int a, int b)
     4  {
     5      int i;
     6      int n1, n2;
     7
     8      i = 0;
     9
    10      n1 = a * b ;
    11      n2 = a - b ;
    12
    13      while (arr[i] > (n1*n2))
    14      {
    15          i = i + 1;
    16      }
    17
    18
    19      return(i);
    20  }
    21

# Intermediate code before and after optimization
# -O gcm for Global Code Motion, -v for verbosity
$ ./mycc.exe -i -O gcm -v test.cm.1.c
TAC Before optimization
```

```
 (0) proc_begin func
 (1) i := 0
 (2) _t0 := a * b
 (3) n1 := _t0
 (4) _t1 := a - b
 (5) n2 := _t1
 (6) label .L0
 (7) _t2 := i * 4
 (8) _t3 := &arr
 (9) _t4 := _t3[_t2]
(10) _t5 := n1 * n2
(11) if _t4 > _t5 goto .L1
(12) goto .L2
(13) label .L1
(14) _t6 := i + 1
(15) i := _t6
(16) goto .L0
(17) label .L2
(18) return i
(19) goto .L3
(20) label .L3
(21) proc_end func

TAC After Local optimization
 (0) proc_begin func
 (1) i := 0
 (2) n1 := a * b
 (3) n2 := a - b
 (4) label .L0
 (5) _t2 := i * 4
 (6) _t3 := &arr
 (7) _t4 := _t3[_t2]
 (8) _t5 := n1 * n2
 (9) if _t4 > _t5 goto .L1
(10) goto .L2
(11) label .L1
(12) i := i + 1
(13) goto .L0
(14) label .L2
(15) return i
(16) goto .L3
(17) label .L3
(18) proc_end func

TAC After (Local and Global) optimization
 (0) proc_begin func
 (1) i := 0
 (2) n1 := a * b
 (3) n2 := a - b
 (4) _t3 := &arr
 (5) _t5 := n1 * n2
 (6) label .L0
 (7) _t2 := i * 4
 (8) _t4 := _t3[_t2]
 (9) if _t4 > _t5 goto .L1
(10) goto .L2
(11) label .L1
```

```
(12) i := i + 1
(13) goto .L0
(14) label .L2
(15) return i
(16) goto .L3
(17) label .L3
(18) proc_end func

# Sample Input C file with Loop invariants that can be moved out
$ cat -n test.cm.2.c
     1  int a[100], b[100];
     2
     3  int transform (int n, int factor)
     4  {
     5      int i;
     6
     7      i = 0;
     8      while (1 == 1)
     9      {
    10          b[i] = a[i] + (factor *2);
    11          i = i + 1;
    12          if (i >= n){
    13              break;
    14          }
    15      }
    16
    17  }
    18

# Intermediate code before and after optimization
$ ./mycc.exe -i -O gcm -v test.cm.2.c
TAC Before optimization
 (0) proc_begin transform
 (1) i := 0
 (2) label .L0
 (3) if 1 == 1 goto .L1
 (4) goto .L4
 (5) label .L1
 (6) _t0 := i * 4
 (7) _t1 := &b
 (8) _t2 := i * 4
 (9) _t3 := &a
(10) _t4 := _t3[_t2]
(11) _t5 := factor * 2
(12) _t6 := _t4 + _t5
(13) _t1[_t0] := _t6
(14) _t7 := i + 1
(15) i := _t7
(16) if i >= n goto .L2
(17) goto .L3
(18) label .L2
(19) goto .L4
(20) label .L3
(21) goto .L0
(22) label .L4
(23) label .L5
(24) proc_end transform
```

```
TAC After Local Optimization
 (0) proc_begin transform
 (1) i := 0
 (2) label .L0
 (3) goto .L1
 (4) goto .L4
 (5) label .L1
 (6) _t0 := i * 4
 (7) _t1 := &b
 (8) _t3 := &a
 (9) _t4 := _t3[_t0]
(10) _t5 := factor * 2
(11) _t6 := _t4 + _t5
(12) _t1[_t0] := _t6
(13) i := i + 1
(14) if i >= n goto .L2
(15) goto .L3
(16) label .L2
(17) goto .L4
(18) label .L3
(19) goto .L0
(20) label .L4
(21) label .L5
(22) proc_end transform

TAC After (Local and Global) optimization
 (0) proc_begin transform
 (1) i := 0
 (2) _t1 := &b
 (3) _t3 := &a
 (4) _t5 := factor * 2
 (5) label .L0
 (6) goto .L1
 (7) goto .L4
 (8) label .L1
 (9) _t0 := i * 4
(10) _t4 := _t3[_t0]
(11) _t6 := _t4 + _t5
(12) _t1[_t0] := _t6
(13) i := i + 1
(14) if i >= n goto .L2
(15) goto .L3
(16) label .L2
(17) goto .L4
(18) label .L3
(19) goto .L0
(20) label .L4
(21) proc_end transform
```

7.3 TARGET CODE OPTIMISATION

In target code optimisation, we look at ways and means of improving the target code generated by the compiler. Similar to the intermediate code optimisation, the main operating principle is that the optimised target code should be correct in all scenarios.

The efficiency of the target code depends on resourceful use of the registers of the processor. Most of the target code optimisation revolves around strategies that can be used during target code generation for better usage of registers (Section 7.3.1). There are other optimisations like the peep-hole optimisation (Section 7.3.2), which improve the efficiency for certain patterns in the target code.

7.3.1 Improved Register Usage

The target code generated by the template-based code generator that we discussed earlier in Chapter 6 leaves scope for optimisation. Let's look at some of the areas where the target code generated by the template-based code generator has scope for improvement.

Consider the Intermediate code and a part of the x86-based target code generated by the template-based code generator in Table 7.75. In the target code generated by the template-based code generator, the moves from register into memory given by 1(c) and the subsequent move from memory to the register given by 2(a) are redundant. The improved target code shown alongside retains the value of '_*t*0' in the register %eax to achieve the same functionality. The improved code is smaller in terms of memory and better in performance.

Table 7.75 *Redundant moves into and from memory*

Source code	**Intermediate code**	**Section of target code from template-based code generator**	**Improved target code**

`1 int a, b, c, d;`
`2`
`3 int func()`
`4 {`
`5     d = (a+b-c);`
`6`
`7     return (d);`
`8 }`
	`(0) proc_begin func`
	`(1) _t0: = a + b`
	`(2) d: = _t0 - c`
	`(3) return d`
	`(4) goto .L0`
	`(5) label .L0`
	`(6) proc_end func`
		`/* _t0: = a + b */`
		`(1a) movl _b, %eax`
		`(1b) addl _a, %eax`
		`(1c) movl %eax, -4(%ebp)`
		
		`/* d := _t0 - c */`
		`(2a) movl -4 (%ebp), %eax`
		`(2b) subl _c, %eax`
		
		`/* return d   */`
		`(3a) movl _d, %eax`
			`/* _t0 := a + b */`
			`(1a) movl _b, %eax`
			`(1b) addl _a, %eax`
			
			`/* d: = _t0 - c */`
			
			`(2b) subl _c, %eax`
			`(2c) movl %eax, _d`
			
			`/* return d */`
			`(3a) movl _d, %eax`

Consider the Intermediate code and a part of the x86-based target code generated by the template-based code generator in Table 7.76. The x86 instructions corresponding to each of the quads contains a load from the memory location to register (e.g. 1a, 2a and 3a) and a move from the register to memory location (e.g. 1c, 2c and 3c). The improved target code retains the results in the registers and uses the less expensive register-to-register instructions (e.g. 2a, 3b shaded in gray) to achieve the same functionality.

Table 7.76 *Using register to register moves*

Source code	**Intermediate code**	**Target code from template-based code generator**	**Improved target code**

`1 int a, b, c, d, e, f;`
`2`
`3 int func()`
`4 {`
`5`
`6   c = a + b;`
`7   e = c + d;`
	`(0) proc_begin func`
	`(1) c: = a + b`
	`(2) e := c + d`
	`(3) f := c + e`
	`(4) return f`
	`(5) goto .L0`
	`(6) label .L0`
		`/* c := a + b */`
		`(1a) movl _b, %eax`
		`(1b) addl _a, %eax`
		`(1c) movl %eax, _c`
		
		`/* e := c + d */`
		`(2a) movl _c, %eax`
			`/* c := a + b */`
			`(1a) movl _b, %eax`
			`(1b) addl _a, %eax`
			`(1c) movl %eax, _c`
			
			`/* e := c + d */`
			`(2a) movl %eax, %edx`

8 f = c + e;	(7) proc_end func	(2b) addl _d, %eax	(2b) addl _d, %edx
9		(2c) movl %eax, _e	(2c) movl %edx, _e
10 return (f);			
11 }		/* f: = c + e */	/* f: = c + e */
		(3a) movl _c, %eax	
		(3b) addl _e, %eax	(3b) addl %edx, %eax
		(3c) movl %eax, _f	(3c) movl %eax, _f
		/* return f */	/* return f */
		(4a) movl _f, %eax	

The target code generated by the template-based code generator does not work optimally in the cases of loops. This is because it does not identify and retain the values of heavily used variables in registers throughout the execution of loop. The performance of loops can be improved manifold by keeping the most heavily used variables in registers during the execution of the loop.

The target code can be improved by replacing the template-based code generator by more sophisticated target code generator that makes better usage of registers. Before we look at the specifics of a particular target code generator, let's study the important issues that need to be addressed commonly by any target code generator in order to make better utilisation of registers.

The assembly instruction using any of the processor registers, as operands are faster than the instructions using memory as operands. To take advantage of this, an efficient target code generator would try and retain program variables in registers for as long as possible, so that it can generate target instructions using registers as operands. The improved target code in Table 7.76 demonstrates the same idea. At the completion of the instruction (1c), the variable '*c*' is contained in register %eax. At (2a), when the value of '*c*' is required, the register %eax is directly used. The value of '*c*' is retained in the register %eax till the quad (3b). At (3b), when the value of '*c*' is needed again, the register %eax is used directly instead of loading from memory. When the number of variables is small in number as in the example of Table 7.76, all the variables computed can be retained in registers. However, when the number of variables is higher than the number of registers available, the target code generators need to identify a smaller sub-set of variables to be retained in memory. The set of variables that would be retained in registers is arrived at by using heuristics like the number of times a variable is used, whether the variable is live, and so on. The process of identifying what variables need to be retained in registers is known as ***register allocation***. Register allocation can be performed at a basic block level or at a global level for the entire procedure. The register allocation performed at a basic block level is called as ***local register allocation (LRA)***. In contrast, the ***global register allocation (GRA)*** aims to allocate the registers across the basic blocks. In local register allocation, it is evident that all the live variables residing in registers need to be saved (or ***spilled***) into memory at the end of the basic block, so that the successor block generates code correctly. In local register allocation, loops cannot be processed optimally, since the registers have to be spilled after every block and it is not possible to retain the most heavily used variables in a register throughout the life of the loop. The register allocation strategy is the most important part of any target code generator aiming to make efficient use of registers.

The target code generators need to track the value of registers and variables during the target code generation in order to make efficient use of registers. The target code generator should have data structures that will help retrieve information like whether the variable's value is present in a register or not, is a register free for allocation to a variable, and so on. The information should be indexed properly in the data structures for quick retrieval and updates. A significant design effort is spent in designing the data structures to track the values of registers and variables in a target generator aspiring to use the registers optimally.

The target code generators need the information on data flow of the program to make informed decisions during target code generation. The data flow analysis of the intermediate code helps in procuring the information. For example, the target code generators identify points when a variable's value needs to be moved from register to memory depending on whether a variable is next used or not. The information about a variable being used next is obtained by data flow analysis of the input program. The target code generators seeking to use the registers in an efficient way need to obtain data flow information of the input program by performing data flow analysis.

We study in detail about a simple x86 target code generator in Section 7.3.1.1 to get a feel of the issues concerning a target code generator intending to make the optimum use of registers.

7.3.1.1 A Simple x86 Target Code Generator In this section, we discuss the features and the implementation of a simple x86 target code generator that makes better use of registers than the template-based code generator. The simple code generator is based on local register allocation, where the register allocation is confined within the basic block. The simple code generator takes the bottom-up approach with regard to register allocation, where it dynamically allocates a register to store the result on a need basis.

The best way to understand the important characteristics of simple target code generator is to analyse its target code for a sample input source. The target code generated by template-based code generator for the same input source allows us to make a good comparison between the two.

Table 7.77 shows an Input source, the corresponding intermediate code, the x86-based target code generated using the template-based code generation as well as the code generated by the simple target code generator.

Table 7.77 *Simple code generation*

(A) **Input Source**	**(B)** **Intermediate Code**	**(C)** **Target Code from Template based Code Generator**	**(D)** **Target Code from Simple Code Generator**

int a,b,c,d,e,f;
 	
int func()
{
	(0) proc_begin func
	(1) c := a + b
	(2) e := c + d
	(3) f := c + e
		/* proc_begin func */
		(0a).align 4
		(0b) .globl _func
		(0c) _func:
			/* proc_begin func */
			(0a).align 4
			(0b) .globl _func
			(0c) _func:

c = a + b;
e = c + d;
f = c + e;
	(4) if f > 100 goto .L0
	
	(5) goto .L1
		(0d) pushl %ebp
		(0e) movl %esp,%ebp
		(0f) subl $16,%esp
			(0d) pushl %ebp
			(0e) movl %esp,%ebp
			(0f) subl $16,%esp

if(f > 100){
f=a+c;
}
	(6) label .L0
	(7) f := a + c
	
	(8) label .L1
		/* c := a + b */
		(1a) movl _b,%eax
		(1b) addl _a,%eax
		(1c) movl %eax,_c
			/* c := a + b */
			(1a) movl _a,%ebx
			(1b) addl _b,%ebx

return(f);
}
	(9) return f
	(10) goto .L2
	
	(11) label .L2
	(12) proc_end func
		/* e := c + d */
		(2a) movl _d,%eax
		(2b) addl _c,%eax
		(2c) movl %eax,_e
			/* e := c + d */
			(2a) movl %ebx,%eax
			(2b) addl _d,%eax

		/* f := c + e */ (3a) movl _e,%eax (3b) addl _c,%eax (3c) movl %eax,_f	/* f := c + e */ (3a) movl %ebx,%edx (3b) addl %eax,%edx

			/*Spilling reg*/ (3c) movl %ebx,_c (3d) movl %eax,_e
		/* if f > 100 goto .L0 */	/* if f > 100 goto .L0*/
			/*Spilling reg*/ (4a) movl %edx,_f
		(4a) movl $100,%eax (4b) cmpl %eax,_f (4c) jg .L0	(4b) movl _f,%ebx (4c) cmpl $100,%ebx (4d) jg .L0
		/* goto .L1 */ (5a) jmp .L1	/* goto .L1 */ (5a) jmp .L1
		/* label .L0 */ (6a) .align 4 (6b) .L0:	/* label .L0 */ (6a) .align 4 (6b) .L0:
		/* f := a + c */ (7a) movl _c,%eax (7b) addl _a,%eax (7c) movl %eax,_f	/* f := a + c */ (7a) movl _a,%ebx (7b) addl _c,%ebx
			/*Spilling reg*/ (7c) movl %ebx,_f
		/* label .L1 */ (8a) .align 4 (8b) .L1:	/* label .L1 */ (8a) .align 4 (8b) .L1:
		/* return f */ (9a) movl _f,%eax	/* return f */ (9a) movl _f,%eax
		/* goto .L2 */ (10a) jmp .L2	/* goto .L2 */ (10a) jmp .L2
		/* label .L2 */ (11a).align 4 (11b) .L2:	/* label .L2 */ (11a).align 4 (11b) .L2:
		/* proc_end func */ (12a) movl %ebp,%esp (12b) popl %ebp (12c) ret	/* proc_end func */ (12a) movl %ebp,%esp (12b) popl %ebp (12c) ret

Each quad is translated into a set of assembly instructions by the simple target code generator, using the information as to which operands are already in registers. In case the operands are already in registers, the registers are directly used in the target code instead of performing a load from memory. The target code instructions (2a), (3a) and (3b) in column (D) are good examples of operands picked up from registers instead of being loaded from memory.

The register to store the result '*x*' of the quad in '*x*: = *y* op *z*' is dynamically allocated at the time of processing the quad. For example, In column (D), at (1a), the register ebx has been allocated to store the variable '*c*', at (2a), the register eax has been allocated to store the variable '*e*'.

The result of the operations (e.g. add/sub/mul, etc.) is left in registers for as long as it is possible before storing it into the memory. For example, after (1b), the value of the variable '*c*' continues to be register ebx. After (2b), the register eax continues to hold '*e*'. The result is stored back into the memory of the variable (spilled) from the register on the triggering of certain conditions like, say, when the variable is no longer used in the basic block or when the register is required for another computation or at the end of basic block, and so on. For example, the instruction (3c) and (3d) spill the values of variables '*c*' and '*e*' since they are no longer used in the basic block.

For the quads having operators such as PROC_BEGIN and PROC_END, the code generated by the template-based approach and simple code generation do not differ. Similarly, for quads using simple operators like LBL, GOTO and CALL, where there is no usage of variables or registers both the approaches yield the same code. The generated target instruction for the quads (0), (5), (6) and (12) in Table 7.77 illustrate the idea.

We study about the simple code generator in detail over the next few sections. The simple code generator needs information about (a) whether a variable is used later in the basic block and (b) whether a variable is live at a given point, while deciding on register spilling as mentioned above. This information is gathered by performing data flow analysis of the input quads for the target code generator.

We discuss the details of the data flow analysis to collect the liveness and next use information at each quad level in the next section. This is followed by a discussion on the data structures and the algorithm used by the simple code generator. The working of the algorithm on a sample set of input quads is presented later in this Section.

Data Flow Analysis for computing Next Use and Liveness We had seen previously how the data flow analysis helped us gather properties like available expressions, reaching definitions, and so on. The simple code generator performs data flow analysis to compute liveness of a variable and another data flow property called the 'next use' at each quad level. These two data flow properties are used during the register spilling.

Consider the quads shown in Table 7.78 in which the variables *a*, *b*, *c*, *d*, *e*, *f* and *g* are present. The quad (1) uses the variables '*b*' and '*c*' and defines a variable '*a*'. The variable '*a*' is ***next used*** after quad (1) in the quad (3). The variable '*c*' is not used in any quad after the quad (1), so there is no next use for '*c*'. We use the notion of next used in the simple code generator while making the register spills.

The idea of ***liveness***, which we studied earlier, is also used in the simple code generation to spill the registers at different points in the block. A variable *v* is said to be live at a point *p*, if it is used in some path in the flow graph starting *p*. To compare the liveness with next use, we can say that the liveness is the next use extending across the blocks. If there is next use for a variable, it is definitely live. A variable might not have next use, but can still be live, if there is a use in some block after the current one.

Table 7.78 *Intermediate code*

```
(0) proc_begin func
(1) a := b + c
(2) f := d + e
(3) g := a + f
(4) f := a - b
(5) d := f + g
(6) label .L0
(7) proc_end func
```

The next uses and liveness information is stored at a quad level for all its operands and used in the simple code generator. Before we look at the algorithms to compute the next uses and liveness information, let's get a feel of how the liveness and next uses information looks like. We assume for the discussion that all the variables '*a*' through '*e*' are all global in scope. Table 7.79 displays the liveness and next uses information for a couple of quads in the sample intermediate code that we saw in Table 7.78.

Table 7.79 *Liveness and next uses information*

Quad	Next use information	Liveness information	Explanation

`(1) a := b + c`	`next_use (1, a) = 3`
	`next_use (1, b) = 4`
	`next_use (1, c) = -1`
		`liveness (1, a) = LIVE`
		`liveness (1, b) = LIVE`
		`liveness (1, c) = LIVE`
			The variable 'a' is next used in quad 3. The variable 'b' is next used in quad 4. The variable 'c' is not used later in this block, hence next_uses (1, c) is -1 to indicate no next use.
			
			The variable 'a' is used later at quad 3, hence it is LIVE. The variable 'b' is used later at 4, hence it is live.
			
			'c' is not used in this block. It is also not defined in any one of the later quads. Since it is a global variable, which can be used in another procedure, it is considered LIVE.

`(2) f := d + e`	`next_use (2, f) = 3`
	`next_use (2, d) = -1`
	`next_use (2, e) = -1`
		`liveness (2, f) = LIVE`
		`liveness (2, d) = DEAD`
		`liveness (2, e) = LIVE`
			The variable 'f' is next used in quad 3. The variable 'd' is not used in quads 3 or 4. It is redefined in quad 5, hence next_use (2, d) is -1. There is no next use for the variable 'e'.
			
			The variable 'f' is used later at quad 3, hence it is LIVE. The variable 'd' is not used, it is redefined in quad 5, hence it is DEAD at this point.
			
			'e' is not used in this block. It is also not defined in any one of the later quads. Since it is a global variable, which can be used in another procedure, it is considered LIVE.

The liveness and next uses information is used for making informed decisions for register spilling and freeing up of registers in the simple code generator. Let's say, for example, the values of *a*, *b* and *c* are in the registers eax, ebx and ecx respectively at the end of the quad 1 in the above example. By knowing that '*c*' is not going to be used again in this block (i.e. next uses is empty), we can choose to free up the register ecx for subsequent generation of code. The fact that '*c*' is live implies that the register ecx needs to be spilled on to the memory for '*c*' before we free up the register for subsequent usage. In the simple code generation strategy, we would try and retain the values of '*a*' and '*b*' in the registers for as long as possible, since there is next usage of these variables.

Let's now look at an algorithm to compute the liveness and next uses at each quad in a block. The idea of the algorithm is to scan backwards starting from the last quad in the block and marking variables for liveness and next use. In the algorithm, we use a temporary table that has an entry for each variable used in the block. For each of the variable, it stores two pieces of Information (a) the current liveness status of the variable—live or dead and (b) the next uses of the variable. The table is initialised with the liveness and next use values at the end of the block, since the quads are processed backwards. For all the variables that are live at the end of the block as indicated by live_OUT set (see Section 7.2.10.3), we initialise the liveness attribute to LIVE in the table. The next uses attribute is initialised to –1, i.e. no next use, for all the variables in the block.

```
/* Initialize a temporary table for next_use and liveness */

/* for each variable 'v' in the block */
for each variable v
{
    tmp_tab[v].next_uses = -1 /* No Next use */

    if (v is in live_OUT of the block){
        tmp_tab[v].liveness = LIVE
    }else{
        tmp_tab[v].liveness = DEAD
    }
}

 /* In a scan backwards from the last quad of the block to first */

for each quad 'res: = arg1 op arg2'
do
    next_uses[quad_no].arg1 = tmp_tab.next_uses[arg1]
    next_uses[quad_no].arg2 = tmp_tab.next_uses[arg2]
    next_uses[quad_no].res  = tmp_tab.next_uses[res]

    liveness[quad_no].arg1 = tmp_tab.liveness[arg1]
    liveness[quad_no].arg2 = tmp_tab.liveness[arg2]
    liveness[quad_no].res  = tmp_tab.liveness[res]

    tmp_tab.liveness[res] = DEAD
    tmp_tab.next_uses[res] = -1  /* No Next use */

    tmp_tab.liveness[arg1] = LIVE
    tmp_tab.next_uses[arg1] = quad_no

    tmp_tab.liveness[arg2] = LIVE
    tmp_tab.next_uses[arg2] = quad_no;
done
```

Algorithm 7.12 *Computing next use and liveness*

The quads are scanned backwards, from the last quad in the block to the first quad. At each quad res := arg1 + arg2, the following steps are performed:

(a) The information available in the temporary table for ‘res’, arg1 and arg2 with respect to liveness and next uses attributes are attached to the quad as given by lines 19 through 25 in Algorithm 7.12.

(b) The entries in the temporary table for variables ‘arg1’, and ‘arg2’ are updated to have liveness attribute as LIVE and next use attribute as the current quad number. The entry in the temporary table for ‘res’ is updated to have the liveness attribute as DEAD, while the next use attribute is marked as –1, signifying no next use. This is given by lines 27 through 34 in Algorithm 7.12.

Let’s watch the algorithm at work on the some of the quads in the intermediate code shown in Table 7.78.

The variables used in the block *a*, *b*, *c*, *d*, *e*, *f* and *g* are all global, and hence they are all live at the end of the block as given by live_OUT set. The temporary table is initialised as shown below:

Var	Liveness	Next Use
a	LIVE	-1
b	LIVE	-1
c	LIVE	-1
d	LIVE	-1
e	LIVE	-1
f	LIVE	-1
g	LIVE	-1

The first quad to be processed is (5) *d* := *f* + *g*. The information from the temporary table above is attached to the quad as mentioned in step (a).

`(5) d := f + g`	`next_uses (5, d) = -1`
	`next_uses (5, f) = -1`
	`next_uses (5, g) = -1`
	
	`liveness (5, d) = LIVE`
	`liveness (5, f) = LIVE`
	`liveness (5, g) = LIVE`

The entries in the temporary table for *d*, *f* and *g* are updated as mentioned in step (b). For the variable ‘*d*’, we make the liveness attribute as DEAD and the next use attribute as –1 (signifying no next use). The variable ‘*f*’ liveness attribute is turned LIVE and its next use is assigned the current quad number 5. The updates in the temporary table are shown in gray below:

Var	Liveness	Next Use
a	LIVE	-1
b	LIVE	-1
c	LIVE	-1
d	DEAD	-1
e	LIVE	-1
f	LIVE	5
g	LIVE	5

The next quad to be processed is (4) *f* := *a* – *b*. The information from the temporary table above is attached to the quad as mentioned in step (a).

(4) f := a - b	next_uses (4, f) = 5 next_uses (4, a) = -1 next_uses (4, b) = -1 liveness (4, f) = LIVE liveness (4, a) = LIVE liveness (4, b) = LIVE

The entries in the temporary table for '*f* ', '*a*' and '*b*' are updated as mentioned in step (b). For the variable '*f* ', we make the liveness attribute as DEAD and the next use attribute as –1 (signifying no next use). The variable '*a*' liveness attribute is turned LIVE and its next use is assigned the current quad number 4. The variable '*b*' liveness attribute is turned LIVE and its next use is assigned the current quad number 4. The updates in the temporary table are shown in gray below:

Var	Liveness	Next Use
a	LIVE	4
b	LIVE	4
c	LIVE	-1
d	DEAD	-1
e	LIVE	-1
f	DEAD	-1
g	LIVE	5

In this way the algorithm continues till the first quad and updates the liveness and next uses information for all the quads. The reader is advised to verify the computation of liveness and next use information using the algorithm with Table 7.80.

Table 7.80 *Next use and liveness information*

Quad	Next Use Information	Liveness information

(1) a := b + c	next_uses (1, a) = {3, 4}
	next_uses (1, b) = {4}
	next_uses (1, c) = { }
		liveness (1, a) = LIVE
		liveness (1, b) = LIVE
		liveness (1, c) = LIVE

(2) f := d + e	next_uses (2, f) = {3}
	next_uses (2, d) = { }
	next_uses (2, e) = { }
		liveness (2, f) = LIVE
		liveness (2, d) = DEAD
		liveness (2, e) = LIVE

(3) g := a + f	next_uses (3, g) = {5}
	next_uses (3, a) = {4}
	next_uses (3, f) = {}
		liveness (3, g) = LIVE
		liveness (3, a) = LIVE
		liveness (3, f) = DEAD

(4) f := a - b	next_uses (4, f) = {5}
	next_uses (4, a) = {}
	next_uses (4, b) = {}
		liveness (4, f) = LIVE
		liveness (4, a) = LIVE
		liveness (4, b) = LIVE

(5) d := f + g	next_uses (5, *d*) = {}
	next_uses (5, *f*) = {}
	next_uses (5, *g*) = {}
		liveness (5, *d*) = LIVE
		liveness (5, *f*) = LIVE
		liveness (5, *g*) = LIVE

Algorithm and Data Structures In this section, we study about the data structures and the algorithm used for implementing the simple code generator. The simple code generator makes use of the liveness and next uses information collected by means of data flow analysis as explained in the last section.

The simple code generator is implemented with the help of two data structures called ***address descriptor table*** and ***register descriptor table***. As the target code is generated quad after quad, these two data structures are consulted to know information like, say, 'Are the operands of the current quad present in a register or memory?', 'Are there any free registers in which the result of the current quad can be stored?' and so on.

The address descriptor table maintains the information as to where the current value of a variable can be found. The current value of a variable can be found in a register or in the memory or a combination of both. Table 7.81 shows a sample address descriptor using x86 registers. It shows 5 variables *p*, *q*, *r*, *s* and *t*. The current value of variable '*p*' is in a register eax. The current value of the variable '*q*' is both register ebx and memory as well. The current value of '*r*' is in memory only. The current value of the variable '*s*' is in two registers ecx and edx. The variable '*t*' is housed in ebx. Observe that both '*q*' and '*t*' are stored in the same register 'ebx'. This is possible after processing a copy statement '*q* = *t*'.

Table 7.81 *Address descriptor table*

Variable name	Current location
p	eax
q	ebx, memory
r	memory
s	ecx, edx
t	ebx

We can see from Table 7.81 that the address descriptor is indexed on the name of the variables (symbol table entries to be precise).

The register descriptor table maintains the information about which variables are currently held in a particular register. A register can hold the values of more than one variable due to copy statements. Table 7.82 shows a sample register descriptor entries using x86 registers. It shows 4 registers eax, ebx, ecx and edx. The register eax holds current value of variable '*p*'. The register ebx holds the current value of the variable '*q*' as well as '*t*'. The current value of the variable '*s*' is in two registers ecx and edx. The register descriptor table is indexed on the register entry. Observe that the register descriptor table shown in Table 7.82 is in sync with the address descriptor shown in Table 7.81, reflecting the same machine state.

Table 7.82 *Register descriptor table*

Register name	Current variables
eax	p
ebx	q, t
ecx	s
edx	s

The algorithm for code generation in simple target code generator consists of 4 steps for each of the quad of the form 'res := arg1 + arg2', where '+' is used to represent any of the IC operators.

1. Identify a register '*r*' in which the result of the quad (res) would be stored. The identification of the register for storing the result is based on simple algorithm described later in this section. At this point, it suffices to know that the algorithm invokes a function get_dst( ) that would return a register name '*r*' in which the result of the quad would be stored.
2. Generate an assembly instruction to move the content of arg1 into '*r*'. In case the value of 'arg1' is in register say '*r*1', then an assembly instruction to move from '*r*1' to '*r*' is generated. In case the value of arg1 is not in register, but only in memory, the assembly instruction to move from memory location arg1 to '*r*' is generated. If it so happens that 'arg1' is already in '*r*', then this instruction need not be generated. The address descriptor table is consulted for knowing if 'arg1' is in a register and fetching the register name.
3. Generate assembly instructions to carry out the operation (op) associated with the quad. For example, if the operation in the quad is say subtract, then assembly instruction 'sub' is generated for an x86 target architecture. The generated target instruction would use 'arg2' as one of the operands and 'arg1' present in register '*r*' as the other operand and store the result in '*r*' itself. In case the value of 'arg2' is in register say '*r*2', then an assembly instruction would use the register '*r*2' instead of using the memory location. The address descriptor table is consulted for knowing if 'arg2' is in a register and fetching the register name.
4. Update the register and address descriptor tables for res, arg1 and arg2.
   - Update the address descriptor table to indicate that the value of 'res' is stored in '*r*' only. Update the register descriptor table to indicate that '*r*' contains the value of the variable 'res' only.
   - If the variable 'arg1' is in a register '*r*1' and arg1 has no next use, then,
     a. if arg1 is LIVE, generate spill code to move the value of *r*1 to memory location of arg1.
     b. Mark the register and address descriptor tables to indicate that the register '*r*1' no longer contains the value of variable 'arg1'. This would allow, get_dst() to pick up the register *r*1 in step (1) in the code generation for the future quads.
   - The same updates as arg1 above are repeated with respect to arg2.

Figure 7.54 illustrates the 4 steps of the target code generation in simple code generator assuming that y and z are not in registers. It also shows the typical actions that happen at each of the 4 steps for a sample IC instruction '$x := y - z$'.

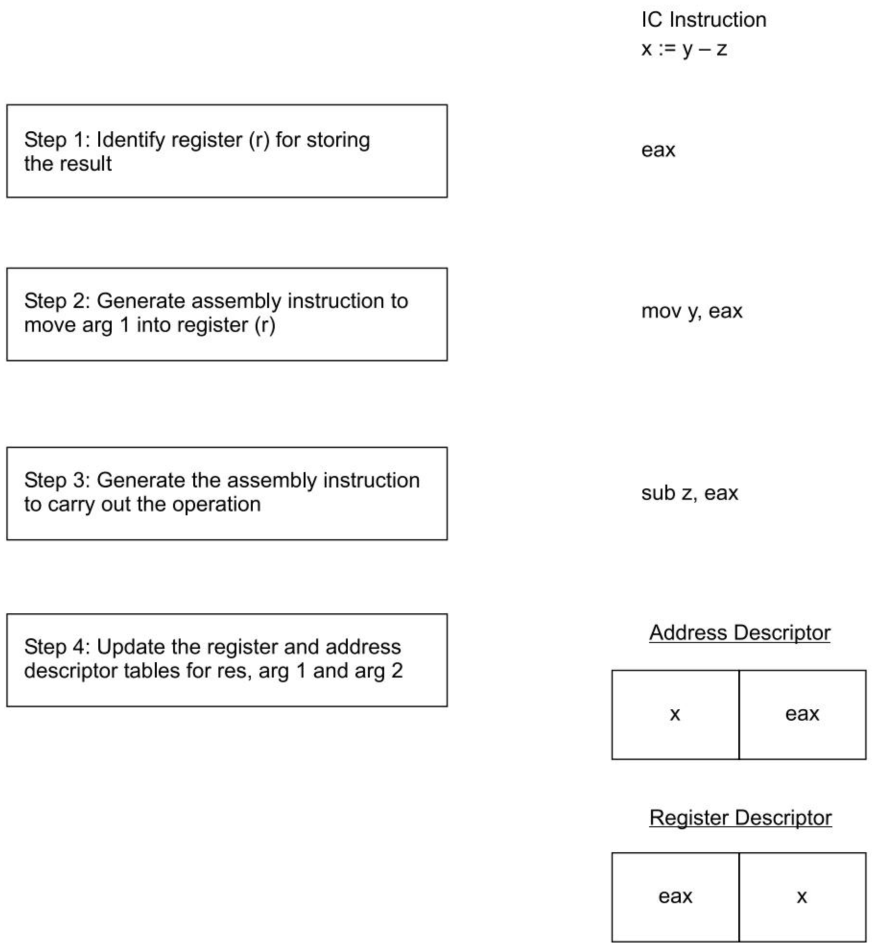

IC Instruction
x := y – z
Step 1: Identify register (r) for storing the result
eax
Step 2: Generate assembly instruction to move arg 1 into register (r)
mov y, eax
Step 3: Generate the assembly instruction to carry out the operation
sub z, eax
Step 4: Update the register and address descriptor tables for res, arg 1 and arg 2
Address Descriptor
x
eax
Register Descriptor
eax
x

Fig. 7.54 *The 4 steps for target code generation in simple code generator*

The function get_dst() is used in step 1 above to identify a register '*r*' for storing the result of the quad. It works on the basis of following algorithm for a given quad 'res: = arg1 + arg2', where '+' is used to represent any of the IC operators.

(a) If 'arg1' is already in register '*r*' and 'arg1' is dead after this statement and '*r*' holds no other variable other than 'arg1', then return '*r*'.

(b) If there is an empty register '*r*' which does not hold the value of any variable, then return '*r*'.

(c) Choose any arbitrary register '*r*'. Let's say the values of variables '*v*1' and '*v*2' are stored in '*r*' at this point in time. We move the contents of the register '*r*' into the memory locations associated with the variables '*v*1' and '*v*2'. We update the address descriptor of '*v*1' and '*v*2' to indicate that '*r*' no longer holds their value. We return the register '*r*'.

This version of get_dst() can be improved by making a more informed choice rather than picking an arbitrary register in (c) above. One approach could be that we could pick a register '*r*' holding the value of variable '*v*' that is used furthest from the current quad.

The other implementation aspects of the simple target generator that are of interest to the reader are detailed below.

- The simple target code generator generates code to spill all the live variables at the end of the block. In cases where the last statement of the block is a GOTO or a conditional GOTO statement, the target code generator spills the live variables before generating code for these or else the spill code would be unreachable. The simple target code generator also generates code to spill all the live variables before a CALL statement, since the called procedure could use the live variables and also overwrite the registers.
- The simple target code generator takes the template-based approach for generating code to the simple operators like PROC_BEGIN, PROC_END, RETURN, RETRIEVE, CALL, LBL, PARAM, GOTO.
- For the operators, which use a specific register '*r*' as a convention, the register '*r*' is spilled in case it is housing a live variable. For example, RETURN uses the register eax on x86 architecture as a convention. The register eax is spilled to the memory, before being used by the return.
- For a quad using the assign operator say *x* := *y*, there are two possible cases
  (a) '*y*' is already in a register say '*r*': There is no code generated if '*y*' is already in a register '*r*'. The register descriptor table is updated to add the variable '*x*' as a part of the set housing '*r*'. The address descriptor table is also updated to reflect the same.
  (b) '*y*' is not present in a register: In the case '*y*' is not already in a register, get_dst( ), fetches a register say '*r*1' to store '*x*'. A 'mov' instruction is generated to move '*y*' from the memory location to the register '*r*1'. The register and address descriptor tables are updated to reflect that '*x*' and '*y*' are housed in '*r*1'.

Illustration of Target Code Generation using Simple Code Generator Let's take the quads shown in Table 7.78 as input and look at how the above-mentioned algorithms of the simple code generator work on it to generate target code.

At the start of target code generation, we initialise the address descriptor table and register descriptor table to indicate that all the values of variables are stored in memory and none of the variables are stored in registers as shown below.

a	Memory
b	Memory
c	Memory
d	Memory
e	Memory
f	Memory
g	Memory

Address descriptor table

eax	None
ebx	None
ecx	None
edx	None

Register descriptor table

The quad, the generated target code and the comments on the working of the algorithm are provided below for all the quads of the example chosen in Table 7.78.

Quad	Generated target code	Comments

(0) proc_begin func	`/* proc_begin func */`
	`   .align 4`
	`.globl _func`
	`_func:`
	`   pushl %ebp`
	`   movl %esp, %ebp`
	`   subl $20, %esp`
		For the operator 'proc_begin' target code is generated based on the template based code generation approach.

Quad	Generated target code	Comments

(1) a := b + c	`/* a: = b + c */`
	`movl _b, %eax`
	`addl _c, %eax`
		Step 1: get_dst() returns a free register 'eax'
		Step 2: Generate target instruction to move the value of 'b' into the selected register eax (mov _b, %eax).
		Step 3: Generate target code to carry out the add operation (addl _c, %eax) and store the result in the selected register eax.
		Step 4: Update the address and register descriptor table to indicate that the register 'eax' contains the value of the variable 'a'.

a	eax
b	Memory
c	Memory
d	Memory
e	Memory
f	Memory
g	Memory

Address descriptor table

eax	a
ebx	None
ecx	None
edx	None

Register descriptor table

Quad	Generated target code	Comments

(2) f := d + e	`/* f : = d + e */`
	`   movl _d, %edx`
	`   addl _e, %edx`
		Step 1: get_dst( )returns a free register 'edx'.
		Step 2 : Generate target instruction to move the value of 'd' into the selected register edx (mov _d, %eax).
		Step 3 : Generate target code to carry out the add operation (addl _e, %edx) and store the result in the register edx.
		Step 4 : Update the address and register descriptor table to indicate that the register 'edx' contains the value of the variable 'f'.

a	eax
b	Memory
c	Memory
d	Memory
e	Memory
f	edx
g	Memory

Address descriptor table

eax	a
ebx	None
ecx	None
edx	f

Register descriptor table

Quad	**Generated target code**	**Comments**

`(3) g := a + f`	`/* g: = a + f */`
	`movl %eax, %ebx`
	`addl %edx, %ebx`
		Step 1: get_dst() returns a free register 'ebx'
		
		Step 2: Generate target instruction to move the value of 'a' into the selected register edx. The value of variable 'a' is already in the register eax as indicated by the address descriptor table. The generated target instruction is mov %eax, %ebx.
		
		Step 3: The other operand 'f' is already in the register 'edx' as indicated by address descriptor table. Generate target code to carry out the add operation (addl %edx, %ebx) and store the result in the register ebx.
		
		Step 4: Update the address and register descriptor table to indicate that the register 'ebx' contains value of the variable 'g'. From the Table 7.80, in the row corresponding to quad 3, we know that the variable 'f' has no next uses and is DEAD after this quad. The register 'edx' in which the value of 'f' is housed at this time is freed up by updating both the desriptor tables.

a	eax
b	Memory
c	Memory
d	Memory
e	Memory
f	–
g	ebx

Address descriptor table

eax	a
ebx	g
ecx	None
edx	None

Register descriptor table

Quad	Generated target code	Comments

`(4) f := a – b`	`/* f: = a – b */`
	`movl %eax, %ecx`
	`subl _b, %ecx`
	
	`/* spilling */`
	`movl %eax, _a`
		Step 1: get_dst(), returns a free register 'ecx.'
		
		Step 2: Generate target instruction to move the value of 'a' into the selected register ecx. The value of variable 'a' is in the register eax as indicated by the address descriptor table. The generated target instruction is mov %eax, %ecx.
		
		Step 3: The other operand 'b' is not available in any register. The third step yields target code to carry out the subtract operation (subl _b, %ecx) and stores the result in the register ecx.
		
		Step 4: Update the address and register descriptor table to indicate that the register 'ecx' contains the value of the variable 'f'. One of the arguments of the quad, the variable 'a' is stored in register eax. It has no next use after this quad, as indicated by the row corresponding to quad 4 in Table 7.80. Since the variable 'a' is LIVE, the register eax is spilled by generating the instruction movl %eax, _a. The register eax is then freed up by updating both the descriptor tables.

a	Memory
b	Memory
c	Memory
d	Memory
e	Memory
f	ecx
g	ebx

Address descriptor table

eax	None
ebx	g
ecx	f
edx	None

Register descriptor table

Quad	Generated target code	Comments

`(5) d := f + g`	`/* d: = f + g */`
	`mov %ecx, %eax`
	`addl %ebx, %eax`
	`/* spilling */`
	`movl %ecx, _f`
	`movl %ebx, _g`
		Step 1: get_dst() returns the free register 'eax.'
		
		Step 2: Generate target instruction to move the value of 'f' into the selected register eax. The value of variable 'f' is already in the register ecx as indicated by the address descriptor table. The generated target instruction is mov %ecx, %eax.
		
		Step 3: The other operand 'g' is available in register %ebx. Generate target code to carry out the add operation (addl %ebx, %eax) and store the result in the register eax.

		Step 4: Update the address and register descriptor table to indicate that the register 'eax' contains the value of the variable 'd'. Both the arguments of the quad, the variable 'f' and 'g' are stored in registers edx and ebx respectively. Both of them have no next use after this quad, but are LIVE as indicated by the row corresponding to quad 5 in Table 7.80. The register edx and ebx are spilled by generating the instructions movl %ecx, _f and movl %ebx, _g. The registers are freed up by updating both the descriptor tables.

a	Memory
b	Memory
c	Memory
d	eax
e	Memory
f	Memory
g	Memory

Address descriptor table

eax	d
ebx	None
ecx	None
edx	None

Register descriptor table

Quad	Generated target code	Comments
(6) proc_end func	/* Spilling reg */ movl %eax, _d /* proc_end func */ movl %ebp, %esp popl %ebp ret	As a part of processing the operators like CALL, comparison operators, goto, proc_end, that change the flow of control, all the live variables that are in registers, but not yet in the memory are spilled. In this case, 'd' is the only live variable that is in a register, but not yet in memory and hence spilled. For the operator 'proc_end' target code is generated based on the template-based code generation approach.

Example 7—Optimised Target Code using Simple Code Generator This section demonstrates the toy C compiler (mycc) generating optimised target code using the simple code generator outlined previously.

The toy C compiler takes as input, a sample C input source and gives out the optimised target code. The dialog below shows 'mycc' taking in some sample input C sources using various operators (like addition, subtraction, etc.) and data structures like arrays, structures, and so on. It generates the corresponding optimised x86 assembly code using the simple code generation strategy. The generated x86 assembly code is assembled to produce an executable binary.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc
```

```
# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$  g++   -DICGEN  -g  -Wall  ic_gen.cc  optimize.cc  target_code_gen.cc  mycc.cc  semantic_
analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Sample Input C file
$ cat -n test.tc.1.c
     1  /*
     2  Simple example
     3  */
     4
     5  int a, b, c, d, e, f, g;
     6
     7  int func()
     8  {
     9      a = b + c;
    10      f =  d + e ;
    11      g = a + f ;
    12      f = a - b ;
    13      d = f + g;
    14  }
    15

# Generating Target code with optimization
# -t for Target Code Optimization
$ ./mycc.exe -t -O all test.tc.1.c
.comm   _a, 4
.comm   _b, 4
.comm   _c, 4
.comm   _d, 4
.comm   _e, 4
.comm   _f, 4
.comm   _g, 4

.text


/* proc_begin func */
       .align 4
.globl _func
_func:
       pushl %ebp
       movl %esp, %ebp
       subl $20, %esp
/* a: = b + c */
       movl _b, %eax
       addl _c, %eax
/* _t1: = d + e */
       movl _d, %ecx
       addl _e, %ecx
/* g := a + _t1 */
```

```
        movl %eax, %edx
        addl %ecx, %edx
/* f: = a - b */
        movl %eax, %ecx
        subl _b, %ecx
/*Spilling Variable*/
        movl %eax, _a
/* d: = f + g */
        movl %ecx, %eax
        addl %edx, %eax
/*Spilling Variable*/
        movl %ecx, _f
/*Spilling Variable*/
        movl %edx, _g
/* proc_end func */
/* Spilling live vars */
/* Vars are
 a  b  c  d  e  f  g */
/*Spilling Variable*/
        movl %eax, _d
        movl %ebp, %esp
        popl %ebp
        ret

# Generate target code with optimization
$ ./mycc.exe -O all -t test.tc.2.c >test.tc.2.s

# Assemble it
$ gcc test.tc.2.s -o test.tc.2.exe

# Execute it
$ ./test.tc.2.exe
BEFORE: x = 0 y = 0 z = 0 q = 0
AFTER: x = 8 y = 12 z = 14 q = -12

# Generate Target code with optimization
$ ./mycc.exe -O all -t test.tc.3.c >test.tc.3.s

# Assemble it
$ gcc test.tc.3.s -o test.tc.3.exe

# Execute it
$ ./test.tc.3.exe
AFTER 1:x = 8 y = -16 ret = -8
AFTER 2:x = 2 y = 16 ret = 18
AFTER 3:x = 25 y = 55 ret = 80

# Generate Target code with optimization
$ ./mycc.exe -O all -t test.tc.4.c >test.tc.4.s

# Assemble it
$ gcc test.tc.4.s -o test.tc.4.exe

# Execute it
$ ./test.tc.4.exe
```

```
BEFORE: The first 5 elements in arr are 0 0 0 0 0
AFTER: The first 5 elements in arr are 0 1 2 3 4

# Generate Target code with optimization
$ ./mycc.exe -O all -t test.tc.5.c >test.tc.5.s

# Assemble it
$ gcc test.tc.5.s -o test.tc.5.exe

# Execute it
$ ./test.tc.5.exe
g_var1 = 200 g_var2 = 25 l_var1 = 5000 l_var2 = 225
```

7.3.1.2 *Limitations of Simple Target Code Generator*

The simple code generator that we studied in the last section is easy to implement and is more efficient than the template-based code generator.

In the simple code generator, the live variables are spilled into memory from the registers at the end of each block. In any of the successor blocks, when one of those live variables is first used, there are instructions to load the variable into a register from the memory. The generated code for register spills at the end of the block and the consequent load instructions into memory in the successor blocks degrades the performance.

The target code generated for loops by the simple code generator is poor. For optimal performance, the most heavily used variables need to be in the registers throughout the life of the loop, across blocks. The simple target code generator cannot afford to keep a variable in a register across blocks, since it operates at a basic block level and spills the live variables at the end of each block.

7.3.2 Peep-hole Optimisation

Peep-hole optimisation is another technique used during the target code optimisation. In peep-hole optimisation, improvements are done local to a small segment of code called the peep-hole or window.

Let's take an example to understand the idea behind peep-hole optimisation. Consider the target code generated for the input source shown in Table 7.83. The target code has been generated using the template-based code generator discussed in Chapter 6.

Table 7.83 *Input source, intermediate code and the target code*

Source code	**Intermediate code**	**Target code from template-based code generator**

`int a, b, c, d;`
 	
`int func()`
`{`
`    d = (a + b - c);`
 	
`    return(d);`
`}`
	`(0) proc_begin func`
	`(1) _t0 := a + b`
	`(2) d := _t0 - c`
	`(3) return d`
	`(4) goto .L0`
	`(5) label .L0`
	`(6) proc_end func`
		`/* _t0 := a + b */`
		`(1a) movl _b, %eax`
		`(1b) addl _a, %eax`
		`(1c) movl %eax, -4(%ebp)`
		
		`/* d: = _t0 - c */`
		`(2a) movl -4(%ebp), %eax`
		`(2b) subl _c, %eax`
		`(2c) movl %eax, _d`
		
		`/* return d */`
		`(3a) movl _d, %eax`

We take the target code generated by the template-based code generator in Table 7.83 and see how the peep-hole optimisation can improve it. Figure 7.55 shows the target code before and after the peep-hole optimisation. The peep-hole optimiser looks at the segment of code (or the peep-hole window) consisting of the x86 assembly instructions labelled (1c) through 2(a) and replaces it with (1c) as shown in Fig. 7.55.

Before optimisation		After optimisation

`(1a) movl _b, %eax`
`(1b) addl _a, %eax`
		`(1a) movl _b, %eax`
		`(1b) addl _a, %eax`

`(1c) movl %eax, -4(%ebp)`
`(2a) movl -4(%ebp), %eax`
	→	`(1c) movl %eax, -4(%ebp)`

`(2b) subl _c, %eax`
`(2c) movl %eax, _d`
`(3a) movl _d, %eax`
		`(2b) subl _c, %eax`
		`(2c) movl %eax, _d`
		`(3a) movl _d, %eax`

Fig. 7.55 *Peep-hole optimisation*

The instructions (1c) and (2a) represent a pattern in the target code of the form mov R, M followed by mov M, R where R is a register and M is a memory. When such a pattern occurs in the target code, the peep-hole optimiser replaces it by a single instruction mov R, M, since the register R already contains the value held in M. A good number of peep-hole optimisations fall in this category, where an identified pattern of instructions in the target code is replaced by a more optimal equivalent set of instructions. Table 7.84 shows some of the other patterns and their replacements by peep-hole optimiser module. Most of these replacements result in either reduction in code size or improvement in speed of execution or both.

Table 7.84 *Pattern and replacement instructions in peep-hole optimisation*

Pattern	**Replacement**	**Comments**

`goto L1`
`L1:`
	`L1:`	The control falls through to the label L1 even without the goto statement. Hence the goto statement can be eliminated.
		This results in reduction of code size.

`addl %ex,1`	`inc %eax`	Use of the machine idiom, the autoincrement operator instead of a explicit add by one. The autoincrement operator takes less cycles to perform the increment. This is an example of using a machine idiom as a replacement instruction. This improves the speed of execution.
`imull %eax, 32`	`lshiftl %eax`	The shift operation involves less cycles than the multiply operation. This is an example of replacement by reduction in strength. This improves the speed of execution.

`imull %eax, 1`
or
`addl %eax, 0`
	None	The multiplication by 1 yields the same value. Hence it can be eliminated.
		The addition by 0 yields the same value. Hence it can be eliminated.
		These are examples of algebraic simplification.
		These improve the speed of execution and also reduce the code size.

Table 7.84 showed cases where the identified pattern is a contiguous set of instructions replaced by another set of optimal set of instructions. It is also possible that the instructions that are analysed for replacement are not in contiguous fashion. For example, Table 7.85 shows a pattern where double jump is avoided by replacing jumps to L1 with L2. The peep-hole optimiser can also be used to eliminate the unnecessary labels for which there are no jumps.

Table 7.85 *Peep-hole optimisation avoiding double jump*

Pattern	Replacement	Comments

`goto L1`
`..`
`..`
`..`
`..`
`goto L1`
`..`
`..`
`..`
`..`
`L1: goto L2`
	`goto L2`
	`..`
	`..`
	`..`
	`..`
	`goto L2`
	`..`
	`..`
	`..`
	`..`
	`L1: goto L2`
		When a label L1 contains a jump to another label L2, then the goto L1 can be replaced by goto L2.
		
		The replaced code is more efficient than the original, since two jumps are avoided at least in some of the cases.

The peep-hole optimisation technique can also be used on the intermediate code to improve it with the same ease.

SUMMARY

An optimising compiler typically has an optimisation phase to improve the intermediate code and the target code in terms of performance and code size.

There are several transformations that can be done on the intermediate code in order to improve the performance like common sub-expression elimination, constant folding, copy propagation, dead code elimination, and so on. Intermediate code optimisation performed within a basic block is known as local optimisation, while the optimisation performed across the basic blocks is termed as global optimisation. Typically compilers perform optimisations at both local and global levels. During local optimisation, the intermediate code is broken into blocks of straight-line code called basic blocks.

A directed acyclic graph (DAG) is a useful data structure for performing local optimisation of the intermediate code. The DAG is used to perform various local intermediate code optimisations like CSE, Dead code elimination, and so on. In order to perform global optimisations in the intermediate code, it is necessary to perform data flow analysis of the input source code. The data flow properties like available expressions, liveness, reaching definitions, etc. are used to perform optimisations like global common sub-expression elimination, global dead code elimination, and so on. A loop-related optimisation in the intermediate code—loop invariant code motion is performed by using a data flow property called as reaching definition.

Most of the target code optimisation involves strategies to have the values of variables in registers and perform operations using the registers instead of memory locations. A simple target code generator, which retains the values of variables in registers for as long as it is possible and uses them for calculations was described in Section 7.3.1.1. Peep-hole optimisation is another commonly employed method to improve the target code.

REVIEW QUESTIONS AND EXERCISES

7.1 What is optimisation? Is there any scope for improving the intermediate code and target code as well?

7.2 What are the common techniques for improving the intermediate code? Explain three of them in detail.

7.3 What is common sub-expression elimination in the context of intermediate code optimisation? Illustrate with an example. Would poorly written code only require common sub-expression elimination to improve the intermediate code?

7.4 What is constant folding in intermediate code optimisation? Illustrate with an example.

7.5 What is copy propagation? Illustrate how the copy propagation facilitates other optimisation opportunities.

7.6 What is dead code elimination? Illustrate with an example.

7.7 What are the common algebraic transformations that can be done for improving the intermediate code?

7.8 What is strength reduction transformation? Illustrate with an example.

7.9 What is a loop invariant code motion optimisation? Illustrate with an example.

7.10 What are induction variables? How does the strength reduction on induction variables help in improving loop optimisation? Illustrate with an example.

7.11 Explain the following terms in the context of intermediate code optimisation (a) basic block (b) directed acyclic graph (c) local and global optimisation.

7.12 What are the main steps in the local optimisation of intermediate code?

7.13 How do you split the intermediate code into basic blocks? Explain with the help of an algorithm.

7.14 Describe an algorithm to construct a DAG from a basic block? Illustrate with an example.

7.15 Explain the algorithm to generate optimised intermediate code by traversing the nodes in a DAG? Illustrate with an example.

7.16 What are the main properties of a DAG? Illustrate those as you construct a DAG.

7.17 What is 'killing' of a DAG node? How does it help in rectifying issues with incorrect optimised intermediate code generation for arrays?

7.18 What are the issues with the optimised intermediate code for pointers while using the basic DAG construction algorithm? What are the corrective measures required in the DAG construction process to eliminate them?

7.19 What are the issues with the optimised intermediate code for procedure calls while using the basic DAG construction algorithm? What are the corrective measures required in the DAG construction process to eliminate them?

7.20 Explain the following terms: (a) flow graph (b) point and path (c) definition and usage of variable in three address code (d) data flow property and data flow analysis.

7.21 What is available expressions? How can it be used to perform global common sub-expression elimination in the intermediate code?

7.22 Explain the terms: (a) generation and killing of expressions (b) universal set of expressions for a basic block of intermediate code? Illustrate these by taking a sample block of three address code.

7.23 Express the relationship between the set of expressions that are available at the beginning of a basic block to the set of expressions that are available at the end of a basic block.

7.24 What is an iterative approach to solving the data flow equations? When do we need it? Give an example in the context of computing available expressions data flow property.

7.25 Given the available expression information, how can you eliminate the re-computation of common sub-expressions at a global level? Illustrate with an example.

7.26 What is liveness of a variable? How can it be used to perform dead code elimination? Illustrate with an example.

7.27 How do you compute (a) live_USES—The set of variables whose use precedes any definition within a basic block (b) live_DEFS – The set of variables whose definition precedes any use within a basic block. Illustrate with a sample three address code.

7.28 Express the relationship between the set of all the variables that are live before reaching the beginning of a basic block to the set of variables that are live at the end of block 'B'?

7.29 Explain the terms (a) domination (b) back edge (c) pre-header (d) natural loop in the context of identifying loops in the intermediate code.

7.30 Describe the algorithm to identify a loop given the back edge. Illustrate with an example.

7.31 What is reaching definitions? How is it used in performing loop invariant code motion optimisation?

7.32 Explain the terms (a) generation of a definition in a block (b) killing of a definition in a block. Illustrate the computation with an example.

7.33 Express the relationship between rd_IN[B], the set of all the definitions reaching the beginning of block 'B' to rd_OUT[B], the set of definitions reaching the end of block 'B'.

7.34 What is use-definition chain? How is it computed from *rd_IN[B]*, the set of all the definitions reaching the beginning of block 'B'?

7.35 What are the steps in performing loop invariant code motion optimisation using the ud-chain information?

7.36 What are the conditions to be satisfied in order to move a TAC statement from within the loop to the pre-header?

7.37 What are the common steps taken by target code generators for producing efficient code?

7.38 Explain any two data flow properties used by target code generators for generating efficient target code.

7.39 Describe an algorithm for computing next use and liveness properties at each quad level in a basic block.

7.40 Describe the data structures and the algorithm for a simple target code generator that retains the values of variables in registers for as long as possible.

7.41 What is peep-hole optimisation? Give five examples of patterns and their replacements used in peep-hole optimisation, justifying the improvement in performance or memory usage.

7.42 State if the following statements are true or false.

(a) The strength reduction transformations identify and replace costly operations by less expensive counterparts.

(b) The DAG is a data structure used for implementing optimising transformations on the intermediate code across basic blocks.

(c) The loop optimisations in the intermediate code are performed during the local optimisation phase.

(d) The order in which DAG nodes are created from the intermediate code during the DAG construction process is in topologically sorted order.

7.43 State if the following statements are true or false.

(a) The leaf nodes in a DAG cannot have any attached identifiers to it.

(b) The input variables to a block manifest as leaves in the DAG during the DAG construction process.
(c) Each node in a flow graph is a quad.
(d) There are $(n - 1)$ points for a basic block containing 'n' quads.

7.44 State if the following statements are true or false.
(a) The 'available expressions' data flow property is used for global common sub-expression elimination.
(b) The 'liveness' data flow property helps in performing global dead code elimination and also in making decisions for retaining a variable's value in a register.
(c) The iterative approach to solving data flow equations is used for resolving the cyclic dependency between the properties of flow graph nodes in the cases of input source having loops.
(d) The dead code elimination can also be performed at a DAG level using live variable information in the form of live_OUT set for the block.

7.45 State if the following statements are true or false.
(a) A node 'd' of a flow graph dominates node 'n', if every path from the initial node to 'n' goes through 'd'.
(b) The dominators[head] containing the tail node in a flow graph detects the presence of a back edge.
(c) A pre-header is a basic block introduced during the loop optimisation to hold the quads that are moved from within the loop.
(d) A ud-chain obtained from reaching definitions analysis is used for performing the loop invariant code motion optimisation in the loops.

INDEX