Systems Programming
and
Operating Systems

Second Revised Edition

Systems Programming and Operating Systems

Second Revised Edition

D M Dhamdhere

Professor and Head
Department of Computer Science & Engineering
Indian Institute of Technology
Mumbai

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices
New Delhi New York St Louis San Francisco Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan Montreal
San Juan Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

A Division of The McGraw-Hill Companies

Information contained in this work has been obtained by Tata McGraw-Hill Publishing Company Limited, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

© 1999, 1993, 1996, Tata McGraw-Hill Publishing Company Limited

Sixth reprint 2001
DZLLCRXLRALCL

No part of this publication can be reproduced in any form or by any means without the prior written permission of the publishers

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited

ISBN 0-07-463579-4

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, printed at
S. P. Printers, Noida 201 301

Dedicated to

Principal Vishwanath Gangadhar Oke
— Professor, philosopher, author
... and a student forever

Preface to the Second Revised Edition

I started work on the second revised edition soon after the second edition was published. The primary motivation was to improve readability and focus, clarity of the fundamental concepts and utility of the examples. This involved a thorough editing of the text, with numerous improvements on each page. Some errors and ambiguities noticed while editing were also corrected.

Apart from these improvements there are plenty of additions to all chapters. The significant ones amongst these are,

- *Evolution of OS Functions (Chapter 9):* A new section on resource allocation and user interface functions.
- *Processes (Chapter 10):* A much enlarged section on threads.
- *Memory Management (Chapter 15):* A new section on the reuse of memory.
- *IO Organization and IO Programming (Chapter 16):* A new section on IO initiation.

I hope the readers will like the new format and compact style. As before, I look forward to comments from the readers.

D M DHAMDHERE

Systems Programming
and
Operating Systems

Second Revised Edition

Preface to the First Edition

This book has grown out of an earlier book *Introduction to System Software* published in 1986, which addressed the recommended curricula in *Systems Programming* (courses 14 of ACM curriculum 68 and CS-11 of ACM curriculum 77), and *Operating Systems and Computer Architecture* (courses CS-6, 7 of ACM curriculum 77 and SE-6, 7 of IEEE curriculum 77). The present book offers a much expanded coverage of the same subject area and also incorporates the recommendations of the ACM-IEEE joint curriculum task force (Computing Curriculum 1991). The contents have also been updated to keep pace with the developments in the field, and the changing emphasis in the teaching of these courses. An example is the way courses titled *Systems Programming* are taught today. As against the 'mostly theory' approach of the previous decade, today the emphasis is on a familiarity with the necessary theory and available software tools. The instructors of these courses have the hard task of finding instructional material on both these aspects. One of the aims of this book is to cater for this requirement through the incorporation of a large number of examples and case studies of the widely used operating systems and software tools available in the field. Treatment of the standard components of system software, viz. assemblers and loaders, is now aimed at the IBM PC. Due to the wide availability of the IBM PC, this makes it possible for students to appreciate the finer aspects in the design of these software components. Case studies of UNIX and UNIX based tools, viz. LEX and YACC, are similarly motivated.

Organization of the book

This book is organized in two parts—*Systems Programming* and *Operating Systems*. The part on Systems Programming introduces the fundamental models of the processing of an HLL program for execution on a computer system, after which separate chapters deal with different kinds of software processors, viz. assemblers, compilers, interpreters and loaders. Each chapter contains examples and case studies so as to offer a comprehensive coverage of the subject matter.

Part II of the book is devoted to an in-depth study of operating systems. The introductory chapter of this part, Chapter 7, identifies the fundamental functions and techniques common to all operating systems. Chapters 8, 9 and 10 offer a detailed treatment of the processor management, storage management and information management functions of the operating systems. These chapters contain motivating discussions, case studies and a set of exercises which would encourage a student to delve deeper into the subject area. Chapter 11 is devoted to an important area of the study of operating systems, that of *concurrent programming*. Evolution of the primitives and contemporary language features for concurrent programming is presented so as to develop an insight into the essentials of

concurrent programming. A case study of a disk manager consolidates the material covered in this chapter. Chapter 12, which is on distributed operating systems, motivates the additional functionalities that de-volve on the operating system due to the distributed environment. This chapter is intended as a primer on distributed operating systems. A detailed treatment has not been possible due to space constraints.

Using this book

This book can be used for the courses on *Systems programming* (or *System software*) at the undergraduate and postgraduate levels, and for an undergraduate course on *Operating systems*. For the former, Part I of the book, together with Chapter 7 from Part II, contains the necessary material. Additionally, parts of Chapter 12 could be used as read-for-yourself material. For a course on *operating systems*, Part II of the book contains the necessary material.

In the courses based on this book, use of concurrently running design-and-implementation projects should be mandatory. Typical project topics would be the development of compilers and interpreters using LEX and YACC, concurrent programming projects, development of OS device drivers, etc.

Apart from use as a text, this book can also be used in the professional computer environment as a reference book or as a text for the enhancement of skills, for new entrants to the field as well as for software managers.

The motivation for writing this book comes from the experience in teaching various courses in the area of system software. I thank all my students for their vital contribution to this book.

DM DHAMDHERE

Contents

Preface to the Second Revised Edition vii
Preface to the Second Edition ix
Preface to the First Edition xi

Part I: SYSTEMS PROGRAMMING

1 Language Processors 1
1.1 Introduction *1*
1.2 Language Processing Activities *5*
1.3 Fundamentals of Language Processing *9*
1.4 Fundamentals of Language Specification *19*
1.5 Language Processor Development Tools *31*
Bibliography 34

2 Data Structures for Language Processing 36
2.1 Search Data Structures *38*
2.2 Allocation Data Structures *52*
Bibliography 57

3 Scanning and Parsing 59
3.1 Scanning *59*
3.2 Parsing *64*
Bibliography 85

4 Assemblers 86
4.1 Elements of Assembly Language Programming *86*
4.2 A Simple Assembly Scheme *91*
4.3 Pass Structure of Assemblers *94*
4.4 Design of a Two Pass Assembler *95*
4.5 A Single Pass Assembler for IBM PC *111*
Bibliography 130

5 Macros and Macro Processors 131
5.1 Macro Definition and Call *132*
5.2 Macro Expansion *133*

5.3 Nested Macro Calls *137*
5.4 Advanced Macro Facilities *138*
5.5 Design of a Macro Preprocessor *145*
Bibliography *161*

6 Compilers and Interpreters **162**
6.1 Aspects of Compilation *162*
6.2 Memory Allocation *165*
6.3 Compilation of Expressions *180*
6.4 Compilation of Control Structures *192*
6.5 Code Optimization *199*
6.6 Interpreters *212*
Bibliography *218*

7 Linkers **221**
7.1 Relocation and Linking Concepts *223*
7.2 Design of a Linker *228*
7.3 Self-Relocating Programs *232*
7.4 A Linker for MS DOS *233*
7.5 Linking for Overlays *245*
7.6 Loaders *248*
Bibliography *248*

8 Software Tools **249**
8.1 Software Tools for Program Development *250*
8.2 Editors *257*
8.3 Debug Monitors *260*
8.4 Programming Environments *262*
8.5 User Interfaces *264*
Bibliography *269*

Part II: OPERATING SYSTEMS

9 Evolution of OS Functions **273**
9.1 OS Functions *273*
9.2 Evolution of OS Functions *276*
9.3 Batch Processing Systems *277*
9.4 Multiprogramming Systems *287*
9.5 Time Sharing Systems *305*
9.6 Real Time Operating Systems *311*
9.7 OS Structure *313*
Bibliography *317*

10 Processes **320**
10.1 Process Definition *320*
10.2 Process Control *322*

10.3 Interacting Processes *327*
10.4 Implementation of Interacting Processes *332*
10.5 Threads *336*
Bibliography *342*

11 Scheduling **343**
11.1 Scheduling Policies *343*
11.2 Job Scheduling *351*
11.3 Process Scheduling *353*
11.4 Process Management in Unix *365*
11.5 Scheduling in Multiprocessor OS *366*
Bibliography *368*

12 Deadlocks **371**
12.1 Definitions *371*
12.2 Resource Status Modelling *372*
12.3 Handling Deadlocks *377*
12.4 Deadlock Detection and Resolution *383*
12.5 Deadlock Avoidance *386*
12.6 Mixed Approach to Deadlock Handling *393*
Bibliography *395*

13 Process Synchronization **396**
13.1 Implementing Control Synchronization *396*
13.2 Critical Sections *399*
13.3 Classical Process Synchronization Problems *408*
13.4 Evolution of Language Features for Process Synchronization *411*
13.5 Semaphores *413*
13.6 Critical Regions *419*
13.7 Conditional Critical Regions *422*
13.8 Monitors *426*
13.9 Concurrent Programming in Ada *437*
Bibliography *443*

14 Interprocess Communication **447**
14.1 Interprocess Messages *447*
14.2 Implementation Issues *448*
14.3 Mailboxes *454*
14.4 Interprocess Messages in Unix *456*
14.5 Interprocess Messages in Mach *458*
Bibliography *459*

15 Memory Management **460**
15.1 Memory Allocation Preliminaries *461*
15.2 Contiguous Memory Allocation *471*

15.3 Noncontiguous Memory Allocation *479*
15.4 Virtual Memory Using Paging *482*
15.5 Virtual Memory Using Segmentation *511*
Bibliography *518*

16 IO Organization and IO Programming **521**
16.1 IO Organization *522*
16.2 IO Devices *526*
16.3 Physical IOCS (PIOCS) *529*
16.4 Fundamental File Organizations *542*
16.5 Advanced IO Programming *544*
16.6 Logical IOCS *552*
16.7 File Processing in Unix *560*
Bibliography *560*

17 File Systems **561**
17.1 Directory Structures *563*
17.2 File Protection *569*
17.3 Allocation of Disk Space *569*
17.4 Implementing File Access *571*
17.5 File Sharing *576*
17.6 File System Reliability *578*
17.7 The Unix File System *584*
Bibliography *587*

18 Protection and Security **588**
18.1 Encryption of Data *588*
18.2 Protection and Security Mechanisms *591*
18.3 Protection of User Files *592*
18.4 Capabilities *596*
Bibliography *603*

19 Distributed Operating Systems **604**
19.1 Definition and Examples *605*
19.2 Design Issues in Distributed Operating Systems *608*
19.3 Networking Issues *611*
19.4 Communication Protocols *615*
19.5 System State and Event Precedence *619*
19.6 Resource Allocation *622*
19.7 Algorithms for Distributed Control *624*
19.8 File Systems *633*
19.9 Reliability *637*
19.10 Security *643*
Bibliography *649*

Index **653**

Preface to the Second Edition

This edition presents a more logical arrangement of topics in Systems Programming and Operating Systems than the first edition. This has been achieved by restructuring the following material into smaller chapters with specific focus:

- *Language processors:* Three new chapters on Overview of language processors, Data structures for language processors, and Scanning and parsing techniques have been added. These are followed by chapters on Assemblers, Macro processors, Compilers and interpreters, and Linkers.
- *Process management:* Process management is structured into chapters on Processes, Scheduling, Deadlocks, Process synchronization, and Interprocess communication.
- *Information management:* Information management is now organized in the form of chapters on IO organization and IO programming, File systems, and Protection and security.

Apart from this, some parts of the text have been completely rewritten and new definitions, examples, figures, sections added and exercises and bibliographies updated. New sections on user interfaces, resource instance and resource request models and distributed control algorithms have been added in the chapters on Software tools, Deadlocks and Distributed operating systems, respectively.

I hope instructors and students will like the new look of the book. Feedback from readers, preferably by email (dmd@cse.iitb.ernet.in), are welcome. I thank my wife and family for their forbearance.

D M DHAMDHERE

Part I

SYSTEMS PROGRAMMING

CHAPTER 1

Language Processors

1.1 INTRODUCTION

Language processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behaviour of a software and the manner in which these ideas are implemented in a computer system. The designer expresses the ideas in terms related to the *application domain* of the software. To implement these ideas, their description has to be interpreted in terms related to the *execution domain* of the computer system. We use the term *semantics* to represent the rules of meaning of a domain, and the term *semantic gap* to represent the difference between the semantics of two domains. Fig. 1.1 depicts the semantic gap between the application and execution domains.

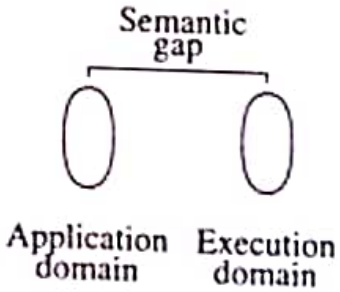


Fig. 1.1 Semantic gap

The semantic gap has many consequences, some of the important ones being large development times, large development efforts, and poor quality of software. These issues are tackled by Software engineering through the use of methodologies and programming languages (PLs). The software engineering steps aimed at the use of a PL can be grouped into

1. Specification, design and coding steps
2. PL implementation steps.

Software implementation using a PL introduces a new domain, the *PL domain*. The semantic gap between the application domain and the execution domain is bridged by the software engineering steps. The first step bridges the gap between the application and PL domains, while the second step bridges the gap between the PL and execution domains. We refer to the gap between the application and PL domains as the *specification-and-design gap* or simply the *specification gap*, and the gap between the PL and execution domains as the *execution gap* (see Fig. 1.2). The specification gap is bridged by the software development team, while the execution gap is bridged by the designer of the programming language processor, viz. a translator or an interpreter.

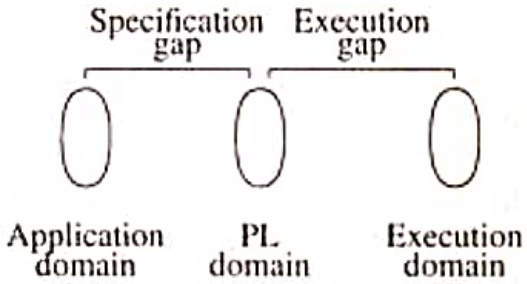


Fig. 1.2 Specification and execution gaps

It is important to note the advantages of introducing the PL domain. The gap to be bridged by the software designer is now between the application domain and the PL domain rather than between the application domain and the execution domain. This reduces the severity of the consequences of semantic gap mentioned earlier. Further, apart from bridging the gap between the PL and execution domains, the language processor provides a diagnostic capability which detects and indicates errors in its input. This helps in improving the quality of the software. (We shall discuss the diagnostic function of language processors in Chapters 3 and 6.)

We define the terms specification gap and execution gap as follows: *Specification gap* is the semantic gap between two specifications of the same task. *Execution gap* is the gap between the semantics of programs (that perform the same task) written in different programming languages. We assume that each domain has a specification language (SL). A specification written in an SL is a *program* in SL. The specification language of the PL domain is the PL itself. The specification language of the execution domain is the machine language of the computer system. We restrict the use of the term execution gap to situations where one of the two specification languages is closer to the machine language of a computer system. In other situations, the term specification gap is more appropriate.

Language processors

Definition 1.1 (Language processor) A *language processor* is a software which bridges a specification or execution gap.

We use the term *language processing* to describe the activity performed by a language processor and assume a diagnostic capability as an implicit part of any form of language processing. We refer to the program form input to a language processor as the *source program* and to its output as the *target program*. The languages in which these programs are written are called *source language* and *target language*, respectively. A language processor typically abandons generation of the target program if it detects errors in the source program.

A spectrum of language processors is defined to meet practical requirements.

1. A *language translator* bridges an execution gap to the machine language (or assembly language) of a computer system. An *assembler* is a language translator whose source language is assembly language. A *compiler* is any language translator which is not an assembler.
2. A *detranslator* bridges the same execution gap as the language translator, but *in the reverse direction*.
3. A *preprocessor* is a language processor which bridges an execution gap but is not a language translator.
4. A *language migrator* bridges the specification gap between two PLs.

Example 1.1 Figure 1.3 shows two language processors. The language processor of part (a) converts a C++ program into a C program, hence it is a preprocessor. The language processor of part (b) is a language translator for C++ since it produces a machine language program. In both cases the source program is in C++. The target programs are the C program and the machine language program, respectively.

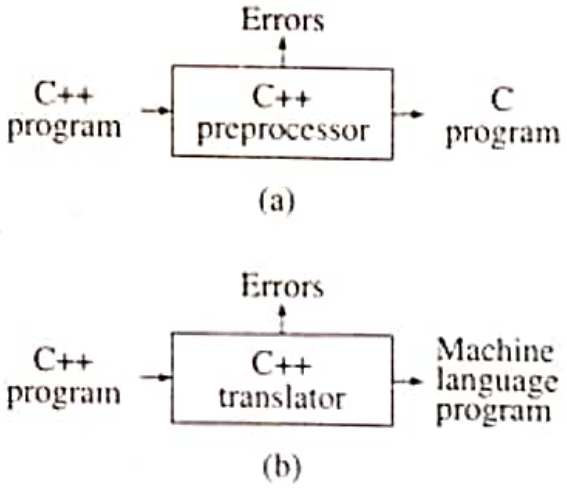


Fig. 1.3 Language processors

Interpreters

An interpreter is a language processor which bridges an execution gap without generating a machine language program. In the classification arising from Definition 1.1,

the interpreter is a language translator. This leads to many similarities between translators and interpreters. From a practical viewpoint many differences also exist between translators and interpreters.

The absence of a target program implies the absence of an output interface of the interpreter. Thus the language processing activities of an interpreter cannot be separated from its program execution activities. Hence we say that an interpreter 'executes' a program written in a PL. In essence, the execution gap vanishes totally. Figure 1.4 is a schematic representation of an interpreter, wherein the interpreter domain encompasses the PL domain as well as the execution domain. Thus, the specification language of the PL domain is identical with the specification language of the interpreter domain. Since the interpreter also incorporates the execution domain, it is as if we have a computer system capable of 'understanding' the programming language. We discuss principles of interpretation in Section 1.2.2.

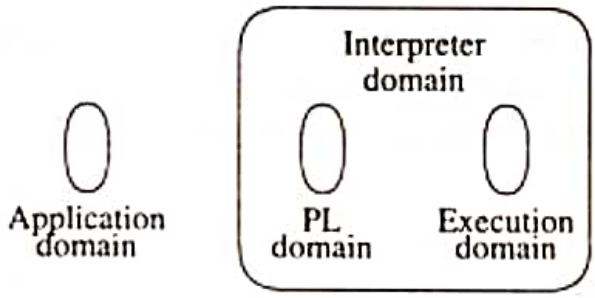


Fig. 1.4 Interpreter

Problem oriented and procedure oriented languages

The three consequences of the semantic gap mentioned at the start of this section are in fact the consequences of a specification gap. Software systems are poor in quality and require large amounts of time and effort to develop due to difficulties in bridging the specification gap. A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain. PL features now directly model aspects of the application domain, which leads to a very small specification gap (see Fig. 1.5). Such PLs can only be used for specific applications, hence they are called *problem oriented languages*. They have large execution gaps, however this is acceptable because the gap is bridged by the translator or interpreter and does not concern the software designer.

A *procedure oriented language* provides general purpose facilities required in most application domains. Such a language is independent of specific application domains and results in a large specification gap which has to be bridged by an application designer.

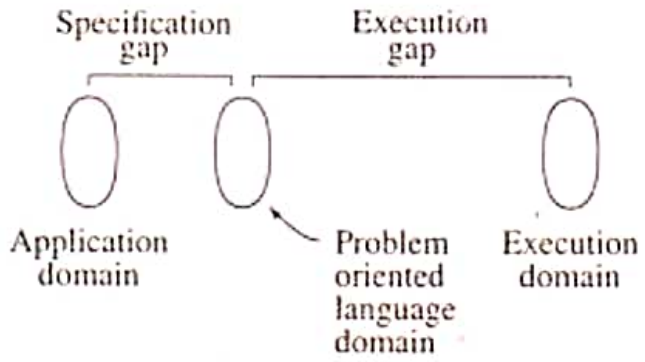


Fig. 1.5 Problem oriented language domain

1.2 LANGUAGE PROCESSING ACTIVITIES

The fundamental language processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap. We name these activities as

1. Program generation activities
2. Program execution activities.

A program generation activity aims at automatic generation of a program. The source language is a specification language of an application domain and the target language is typically a procedure oriented PL. A program execution activity organizes the execution of a program written in a PL on a computer system. Its source language could be a procedure oriented language or a problem oriented language.

1.2.1 Program Generation

Figure 1.6 depicts the program generation activity. The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL. In effect, the program generator introduces a new domain between the application and PL domains (see Fig. 1.7). We call this the *program generator domain*. The specification gap is now the gap between the application domain and the program generator domain. This gap is smaller than the gap between the application domain and the target PL domain.

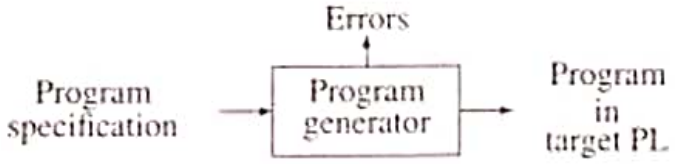


Fig. 1.6 Program generation

Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.

Fig. 1.6 Program generation

Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.

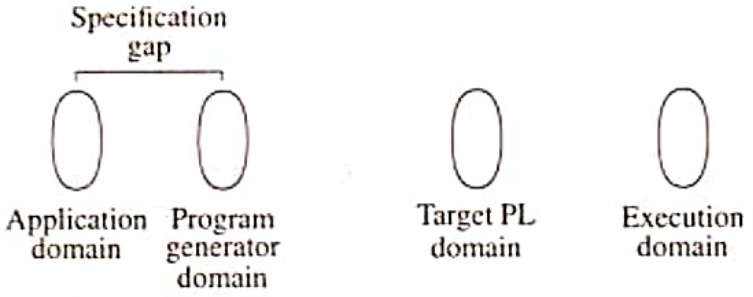


Fig. 1.7 Program generator domain

The harder task of bridging the gap to the PL domain is performed by the generator. This arrangement also reduces the testing effort. Proving the correctness of the program generator amounts to proving the correctness of the transformation of Fig. 1.6. This would be performed while implementing the generator. To test an application generated by using the generator, it is necessary to only verify the correctness of the specification input to the program generator. This is a much simpler task than verifying correctness of the generated program. This task can be further simplified by providing a good diagnostic (i.e. error indication) capability in the program generator which would detect inconsistencies in the specification.

It is more economical to develop a program generator than to develop a problem oriented language. This is because a problem oriented language suffers a very large execution gap between the PL domain and the execution domain (see Fig. 1.5), whereas the program generator has a smaller semantic gap to the target PL domain, which is the domain of a standard procedure oriented language. The execution gap between the target PL domain and the execution domain is bridged by the compiler or interpreter for the PL.

Example 1.2 A screen handling program (also called a form fillin program) handles screen IO in a data entry environment. It displays the field headings and default values for various fields in the screen and accepts data values for the fields. Figure 1.8 shows a screen for data entry of employee information. A data entry operator can move the cursor to a field and key in its value. The screen handling program accepts the value and stores it in a data base.

A screen generator generates screen handling programs. It accepts a specification of the screen to be generated (we will call it the screen spec) and generates a program that performs the desired screen handling. The specification for some fields in Fig. 1.8 could be as follows:

```
Employee name   :  char : start(line=2,position=25)
                   end(line=2,position=80)
Married         :  char : start(line=10,position=25)
                   end(line=10,position=27)
                   default('Yes')
```

Errors in the specification, e.g. invalid start or end positions or conflicting specifications for a field, are detected by the generator. The generated screen handling program

validates the data during data entry, e.g. the *age* field must only contain digits, the *sex* field must only contain M or F, etc.

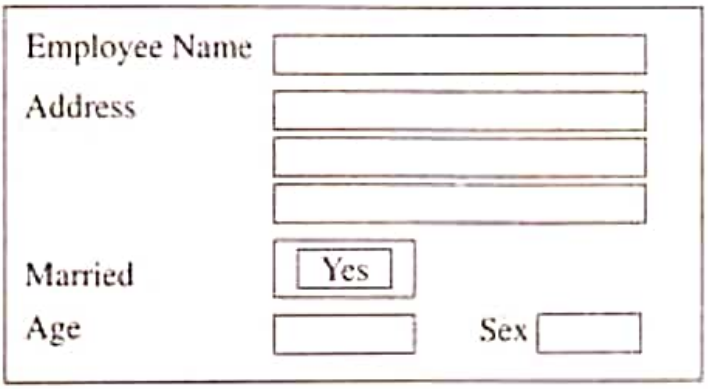


Fig. 1.8 Screen displayed by a screen handling program

1.2.2 Program Execution

Two popular models for program execution are translation and interpretation.

Program translation

The program translation model bridges the execution gap by translating a program written in a PL, called the *source program* (SP), into an equivalent program in the machine or assembly language of the computer system, called the *target program* (TP) (see Fig. 1.9).

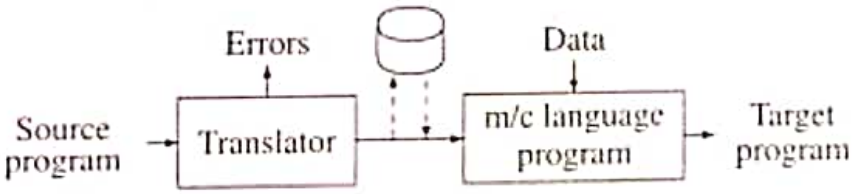


Fig. 1.9 Program translation model

Characteristics of the program translation model are:

- A program must be translated before it can be executed.
- The translated program may be saved in a file. The saved program may be executed repeatedly.
- A program must be retranslated following modifications.

Program interpretation

Figure 1.10(a) shows a schematic of program interpretation. The interpreter reads the source program and stores it in its memory. During interpretation it takes a source

statement, determines its meaning and performs actions which implement it. This includes computational and input-output actions.

To understand the functioning of an interpreter, note the striking similarity between the interpretation schematic (Fig. 1.10(a)) and a schematic of the execution of a machine language program by the CPU of a computer system (Fig. 1.10(b)). The CPU uses a *program counter* (PC) to note the address of the next instruction to be executed. This instruction is subjected to the *instruction execution cycle* consisting of the following steps:

1. Fetch the instruction.
2. Decode the instruction to determine the operation to be performed, and also its operands.
3. Execute the instruction.

At the end of the cycle, the instruction address in PC is updated and the cycle is repeated for the next instruction. Program interpretation can proceed in an analogous manner. Thus, the PC can indicate which statement of the source program is to be interpreted next. This statement would be subjected to the *interpretation cycle*, which could consist of the following steps:

1. Fetch the statement.
2. Analyse the statement and determine its meaning, viz. the computation to be performed and its operands.
3. Execute the meaning of the statement.

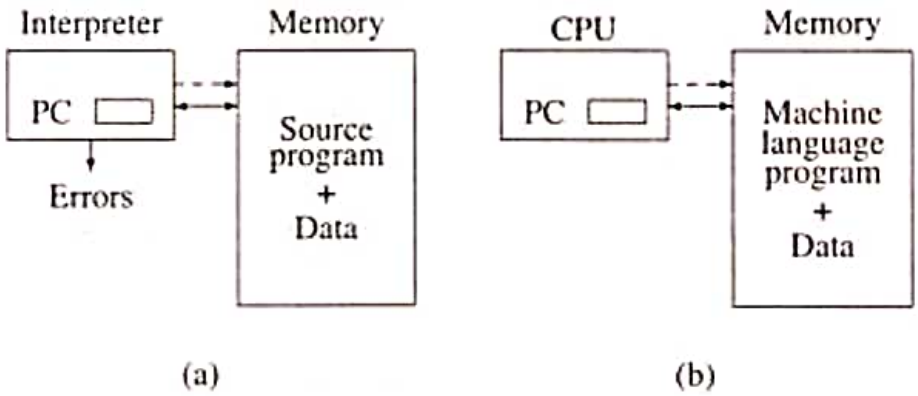


Fig. 1.10 Schematics of (a) interpretation, (b) program execution

From this analogy, we can identify the following characteristics of interpretation:

- The source program is retained in the source form itself, i.e. no target program form exists,
- A statement is analysed during its interpretation.

Section 6.6 contains a detailed description of interpretation.

Comparison

A fixed cost (the translation overhead) is incurred in the use of the program translation model. If the source program is modified, the translation cost must be incurred again irrespective of the size of the modification. However, execution of the target program is efficient since the target program is in the machine language. Use of the interpretation model does not incur the translation overheads. This is advantageous if a program is modified between executions, as in program testing and debugging. Interpretation is however slower than execution of a machine language program because of Step 2 in the interpretation cycle.

1.3 FUNDAMENTALS OF LANGUAGE PROCESSING

Definition 1.2 (Language Processing)

Language Processing ≡ *Analysis of SP + Synthesis of TP.*

Definition 1.2 motivates a generic model of language processing activities. We refer to the collection of language processor components engaged in analysing a source program as the *analysis phase* of the language processor. Components engaged in synthesizing a target program constitute the *synthesis phase*.

A specification of the source language forms the basis of source program analysis. The specification consists of three components:

1. *Lexical rules* which govern the formation of valid lexical units in the source language.
2. *Syntax rules* which govern the formation of valid statements in the source language.
3. *Semantic rules* which associate meaning with valid statements of the language.

The analysis phase uses each component of the source language specification to determine relevant information concerning a statement in the source program. Thus, analysis of a source statement consists of lexical, syntax and semantic analysis.

Example 1.3 Consider the statement

```
percent_profit := (profit * 100) / cost_price;
```

in some programming language. Lexical analysis identifies `:=`, `*` and `/` as operators, 100 as a constant and the remaining strings as identifiers. Syntax analysis identifies the statement as an assignment statement with `percent_profit` as the left hand side and `(profit * 100) / cost_price` as the expression on the right hand side. Semantic analysis determines the meaning of the statement to be the assignment of

$$\frac{\texttt{profit} \times 100}{\texttt{cost_price}}$$

to `percent_profit`.

The synthesis phase is concerned with the construction of target language statement(s) which have the same meaning as a source statement. Typically, this consists of two main activities:

- Creation of data structures in the target program
- Generation of target code.

We refer to these activities as *memory allocation* and *code generation*, respectively.

Example 1.4 A language processor generates the following assembly language statements for the source statement of Ex. 1.3.

```
                    MOVER       AREG, PROFIT
                    MULT        AREG, 100
                    DIV         AREG, COST_PRICE
                    MOVEM       AREG, PERCENT_PROFIT
                    ...
PERCENT_PROFIT      DW          1
PROFIT              DW          1
COST_PRICE          DW          1
```

where MOVER and MOVEM move a value from a memory location to a CPU register and vice versa, respectively, and DW reserves one or more words in memory. Needless to say, both memory allocation and code generation are influenced by the target machine's architecture.

Phases and passes of a language processor

From the preceding discussion it is clear that a language processor consists of two distinct phases—the analysis phase and the synthesis phase. Figure 1.11 shows a schematic of a language processor. This schematic, as also Examples 1.3 and 1.4 may give the impression that language processing can be performed on a statement-by-statement basis—that is, analysis of a source statement can be immediately followed by synthesis of equivalent target statements. This may not be feasible due to:

- Forward references

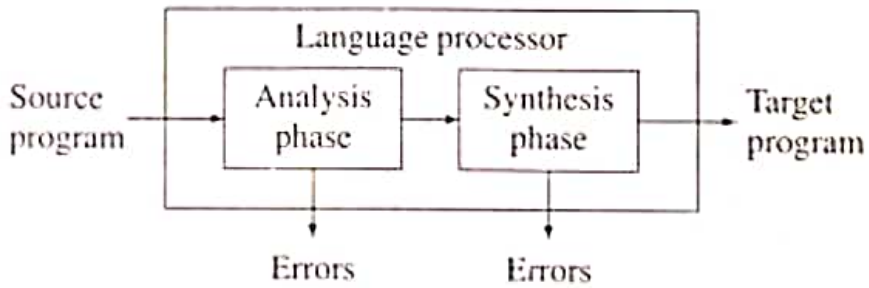


Fig. 1.11 Phases of a language processor

- Issues concerning memory requirements and organization of a language processor.

We discuss these issues in the following.

Definition 1.3 (Forward reference) *A* forward reference *of a program entity is a reference to the entity which precedes its definition in the program.*

While processing a statement containing a forward reference, a language processor does not possess all relevant information concerning the referenced entity. This creates difficulties in synthesizing the equivalent target statements. This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available. Postponing the generation of target code may also reduce memory requirements of the language processor and simplify its organization.

Example 1.5 Consider the statement of Ex. 1.3 to be a part of the following program in some programming language:

```
percent_profit := (profit * 100) / cost_price;
...
long profit;
```

The statement `long profit;` declares `profit` to have a double precision value. The reference to `profit` in the assignment statement constitutes a forward reference because the declaration of `profit` occurs later in the program. Since the type of `profit` is not known while processing the assignment statement, correct code cannot be generated for it in a statement-by-statement manner.

Departure from the statement-by-statement application of Definition 1.2 leads to the *multipass model* of language processing.

Definition 1.4 (Language processor pass) *A* language processor pass *is the processing of every statement in a source program, or its equivalent representation, to perform a language processing function (a set of language processing functions).*

Here 'pass' is an abstract noun describing the processing performed by the language processor. For simplicity, the part of the language processor which performs one pass over the source program is also called a pass.

Example 1.6 It is possible to process the program fragment of Ex. 1.5 in two passes as follows:

Pass I	:	Perform analysis of the source program and note relevant information
Pass II	:	Perform synthesis of target program

Information concerning the type of `profit` is noted in pass I. This information is used during pass II to perform code generation.

Intermediate representation of programs

The language processor of Ex. 1.6 performs certain processing more than once. In pass I, it analyses the source program to note the type information. In pass II, it once again analyses the source program to generate target code using the type information noted in pass I. This can be avoided using an *intermediate representation* of the source program.

Definition 1.5 (Intermediate Representation (IR)) *An* intermediate representation (IR) *is a representation of a source program which reflects the effect of some, but not all, analysis and synthesis tasks performed during language processing.*

The IR is the 'equivalent representation' mentioned in Definition 1.4. Note that the words *'but not all'* in Definition 1.5 differentiate between the target program and an IR. Figure 1.12 depicts the schematic of a two pass language processor. The first pass performs analysis of the source program and reflects its results in the intermediate representation. The second pass reads and analyses the IR, instead of the source program, to perform synthesis of the target program. This avoids repeated processing of the source program. The first pass is concerned exclusively with source language issues. Hence it is called the *front end* of the language processor. The second pass is concerned with program synthesis for a specific target language. Hence it is called the *back end* of the language processor. Note that the front and back ends of a language processor need not coexist in memory. This reduces the memory requirements of a language processor.

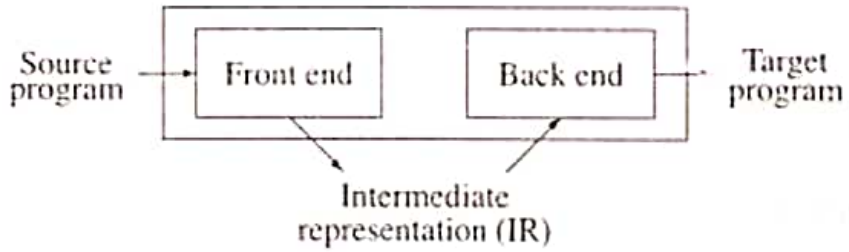


Fig. 1.12 Two pass schematic for language processing

Desirable properties of an IR are:

- *Ease of use:* IR should be easy to construct and analyse.
- *Processing efficiency:* efficient algorithms must exist for constructing and analysing the IR.
- *Memory efficiency:* IR must be compact.

Like the pass structure of language processors, the nature of intermediate representation is influenced by many design and implementation considerations. In the following sections we will focus on the fundamental issues in language processing. Wherever possible and relevant, we will comment on suitable IR forms.

Semantic actions

As seen in the preceding discussions, the front end of a language processor analyses the source program and constructs an IR. All actions performed by the front end, except lexical and syntax analysis, are called *semantic actions*. These include actions for the following:

1. Checking semantic validity of constructs in SP
2. Determining the meaning of SP
3. Constructing an IR.

1.3.1 A Toy Compiler

We briefly describe the front end and back end of a toy compiler for a Pascal-like language.

1.3.1.1 The Front End

The front end performs lexical, syntax and semantic analysis of the source program. Each kind of analysis involves the following functions:

1. Determine validity of a source statement from the viewpoint of the analysis.
2. Determine the 'content' of a source statement.
3. Construct a suitable representation of the source statement for use by subsequent analysis functions, or by the synthesis phase of the language processor.

The word 'content' has different connotations in lexical, syntax and semantic analysis. In lexical analysis, the content is the lexical class to which each lexical unit belongs, while in syntax analysis it is the syntactic structure of a source statement. In semantic analysis the content is the meaning of a statement—for a declaration statement, it is the set of attributes of a declared variable (e.g. type, length and dimensionality), while for an imperative statement, it is the sequence of actions implied by the statement.

Each analysis represents the 'content' of a source statement in the form of (1) tables of information, and (2) description of the source statement. Subsequent analysis uses this information for its own purposes and either adds information to these tables and descriptions, or constructs its own tables and descriptions. For example, syntax analysis uses information concerning the lexical class of lexical units and constructs a representation for the syntactic structure of the source statement. Semantic analysis uses information concerning the syntactic structure and constructs a representation for the meaning of the statement. The tables and descriptions at the end of semantic analysis form the IR of the front end (see Fig. 1.12).

Output of the front end

The IR produced by the front end consists of two components:

1. Tables of information
2. An *intermediate code* (IC) which is a description of the source program.

Tables

Tables contain the information obtained during different analyses of SP. The most important table is the symbol table which contains information concerning all identifiers used in the SP. The symbol table is built during lexical analysis. Semantic analysis adds information concerning symbol attributes while processing declaration statements. It may also add new names designating temporary results.

Intermediate code (IC)

The IC is a sequence of IC units, each IC unit representing the meaning of one action in SP. IC units may contain references to the information in various tables.

Example 1.7 Figure 1.13 shows the IR produced by the analysis phase for the program

```
i : integer;
a,b : real;
a := b+i;
```

Symbol table

	symbol	*type*	*length*	*address*
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

Intermediate code

1. Convert (Id, #1) to real, giving (Id, #4)
2. Add (Id, #4) to (Id, #3), giving (Id, #5)
3. Store (Id, #5) in (Id, #2)

Fig. 1.13 IR for the program of Example 1.8

The symbol table contains information concerning the identifiers and their types. This information is determined during lexical and semantic analysis, respectively. In IC, the specification (Id, #1) refers to the id occupying the first entry in the table. Note that `i*` and `temp` are temporary names added during semantic analysis of the assignment statement.

Lexical analysis (Scanning)

Lexical analysis identifies the lexical units in a source statement. It then classifies the units into different lexical classes, e.g. id's, constants, reserved id's, etc. and

enters them into different tables. This classification may be based on the nature of a string or on the specification of the source language. (For example, while an integer constant is a string of digits with an optional sign, a reserved id is an id whose name matches one of the reserved names mentioned in the language specification.) Lexical analysis builds a descriptor, called a *token*, for each lexical unit. A token contains two fields—*class code*, and *number in class*. *class code* identifies the class to which a lexical unit belongs. *number in class* is the entry number of the lexical unit in the relevant table. We depict a token as [*Code #no*], e.g. [*Id #10*]. The IC for a statement is thus a string of tokens.

Example 1.8 The statement `a := b+i;` is represented as the string of tokens

[*Id #2*] [*Op #5*] [*Id #3*] [*Op #3*] [*Id #1*] [*Op #10*]

where [*Id #2*] stands for 'identifier occupying entry #2 in the Symbol table', i.e. the id `a` (see Fig. 1.13). [*Op #5*] similarly stands for the operator ':=', etc.

Syntax analysis (Parsing)

Syntax analysis processes the string of tokens built by lexical analysis to determine the statement class, e.g. assignment statement, if statement, etc. It then builds an IC which represents the structure of the statement. The IC is passed to semantic analysis to determine the meaning of the statement.

Example 1.9 Figure 1.14 shows IC for the statements `a,b : real;` and `a := b+i;`. A tree form is chosen for IC because a tree can represent the hierarchical structure of a PL statement appropriately. Each node in a tree is labelled by an entity. For simplicity, we use the source form of an entity, rather than its token. IC for the assignment statement shows that the computation `b+i` is a part of the expression occurring on the RHS of the assignment.

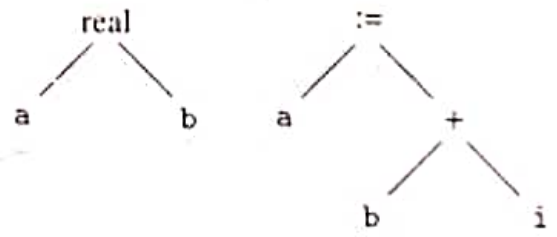


Fig. 1.14 IC for the statements `a,b : real; a := b+i;`

Semantic analysis

Semantic analysis of declaration statements differs from the semantic analysis of imperative statements. The former results in addition of information to the symbol table, e.g. type, length and dimensionality of variables. The latter identifies the sequence of actions necessary to implement the meaning of a source statement. In both cases the structure of a source statement guides the application of the semantic rules. When semantic analysis determines the meaning of a subtree in the IC, it adds

information to a table or adds an action to the sequence of actions. It then modifies the IC to enable further semantic analysis. The analysis ends when the tree has been completely processed. The updated tables and the sequence of actions constitute the IR produced by the analysis phase.

Example 1.10 Semantic analysis of the statement `a := b+i;` proceeds as follows:

1. Information concerning the type of the operands is added to the IC tree. The IC tree now looks as in Fig. 1.15(a).
2. Rules of meaning governing an assignment statement indicate that the expression on the right hand side should be evaluated first. Hence focus shifts to the right subtree rooted at '+'.
3. Rules of addition indicate that type conversion of `i` should be performed to ensure type compatibility of the operands of '+'. This leads to the action

   (i) Convert `i` to real, giving `i*`.

   which is added to the sequence of actions. The IC tree under consideration is modified to represent the effect of this action (see Fig. 1.15(b)). The symbol `i*` is now added to the symbol table.

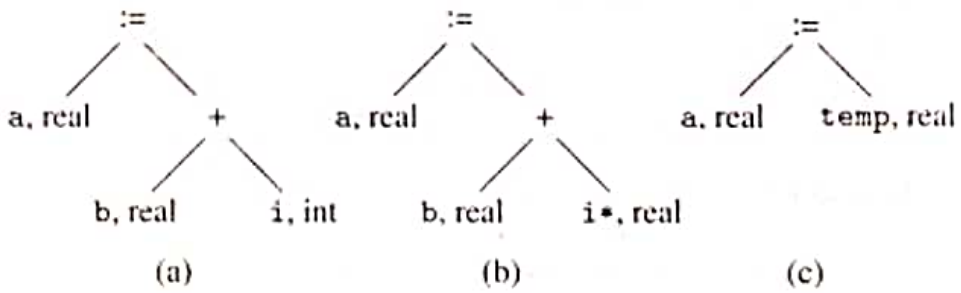


Fig. 1.15 Steps in semantic analysis of an assignment statement

4. Rules of addition indicate that the addition is now feasible. This leads to the action

   (ii) Add `i*` to `b`, giving `temp`.

   The IC tree is transformed as shown in Fig. 1.15(c), and `temp` is added to the symbol table.
5. The assignment can be performed now. This leads to the action

   (iii) Store `temp` in `a`.

This completes semantic analysis of the statement. Note that IC generated here is identical with that shown in Fig. 1.13.

Figure 1.16 shows the schematic of the front end where arrows indicate flow of data.

1.3.1.2 The Back End

The back end performs memory allocation and code generation.

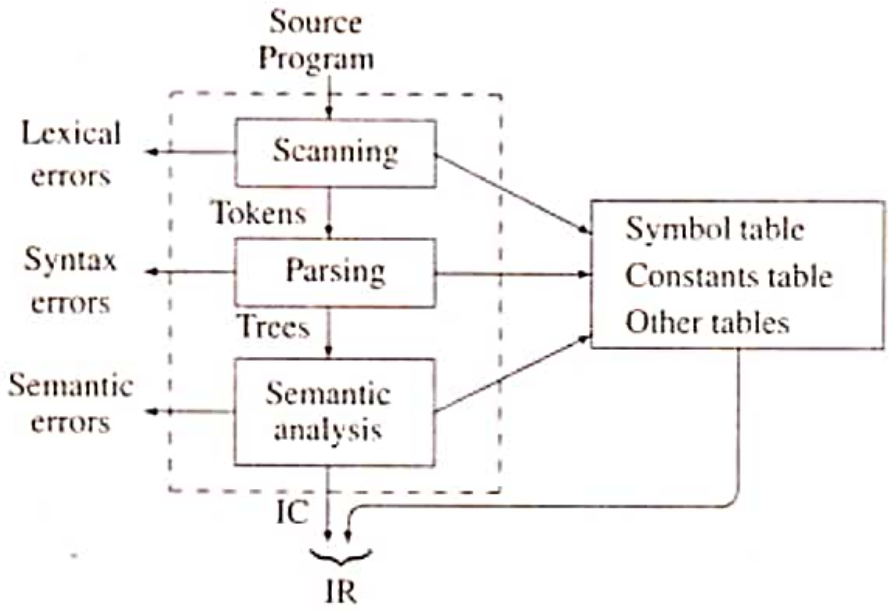


Fig. 1.16 Front end of the toy compiler

Memory allocation

Memory allocation is a simple task given the presence of the symbol table. The memory requirement of an identifier is computed from its type, length and dimensionality, and memory is allocated to it. The address of the memory area is entered in the symbol table.

Example 1.11 After memory allocation, the symbol table looks as shown in Fig. 1.17. The entries for `i*` and `temp` are not shown because memory allocation is not needed for these id's.

	symbol	*type*	*length*	*address*
1	i	int		2000
2	a	real		2001
3	b	real		2002

Fig. 1.17 Symbol table after memory allocation

Note that certain decisions have to precede memory allocation, for example, whether `i*` and `temp` of Ex. 1.10 should be allocated memory. These decisions are taken in the preparatory steps of code generation.

Code generation

Code generation uses knowledge of the target architecture, viz. knowledge of instructions and addressing modes in the target computer, to select the appropriate instructions. The important issues in code generation are:

1. Determine the places where the intermediate results should be kept, i.e. whether they should be kept in memory locations or held in machine registers. This is a preparatory step for code generation.
2. Determine which instructions should be used for type conversion operations.
3. Determine which addressing modes should be used for accessing variables.

Example 1.12 For the sequence of actions for the assignment statement `a := b+i;` in Ex. 1.10, viz.

(i) Convert `i` to real, giving `i*`,
(ii) Add `i*` to `b`, giving `temp`.
(iii) Store `temp` in `a`.

the synthesis phase may decide to hold the values of `i*` and `temp` in machine registers and may generate the assembly code

```
CONV_R     AREG, I
ADD_R      AREG, B
MOVEM      AREG, A
```

where `CONV_R` converts the value of `I` into the real representation and leaves the result in `AREG`. `ADD_R` performs the addition in real mode and `MOVEM` puts the result into the memory area allocated to `A`.

Some issues involved in code generation may require the designer to look beyond machine architecture. For example, whether or not the value of `temp` should be stored in a memory location in Ex. 1.12 would partly depend on whether the value of `b+i` is used more than once in the program. This is an aspect of code optimization.

Figure 1.18 shows a schematic of the back end.

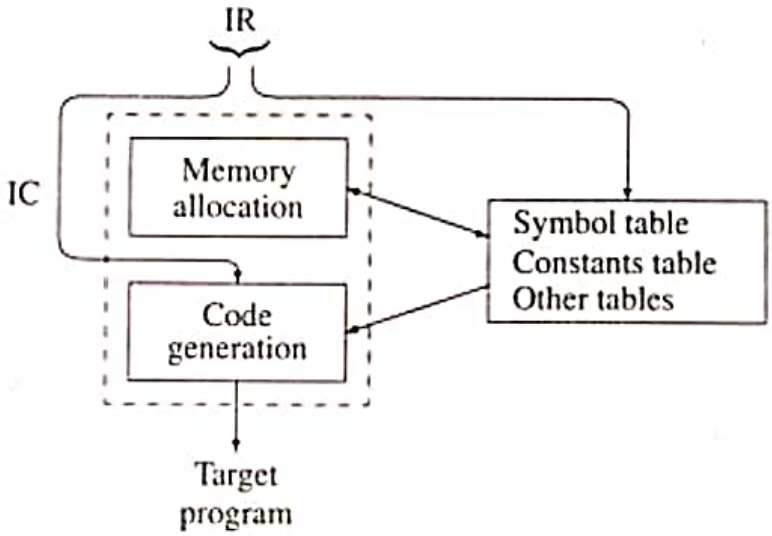


Fig. 1.18 Back end of the toy compiler

1.4 FUNDAMENTALS OF LANGUAGE SPECIFICATION

As mentioned earlier, a specification of the source language forms the basis of source program analysis. In this section, we shall discuss important lexical, syntactic and semantic features of a programming language.

1.4.1 Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. This section discusses key concepts and notions from formal language grammars. A language L can be considered to be a collection of valid sentences. Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as a *formal language*. A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal symbols, alphabet and strings

The *alphabet* of L, denoted by the Greek symbol Σ, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols in Σ. A symbol in the alphabet is known as a *terminal symbol* (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g.

$$\Sigma \equiv \{ \mathtt{a}, \mathtt{b}, \ldots \mathtt{z}, \mathtt{0}, \mathtt{1}, \ldots \mathtt{9} \}$$

Here the symbols {, ',' and } are part of the notation. We call them *metasymbols* to differentiate them from terminal symbols. Throughout this discussion we assume that metasymbols are distinct from the terminal symbols. If this is not the case, i.e. if a terminal symbol and a metasymbol are identical, we enclose the terminal symbol in quotes to differentiate it from the metasymbol. For example, the set of punctuation symbols of English can be defined as

$$\{ \mathtt{:}, \mathtt{;}, \text{`,'}, \ldots \}$$

where ',' denotes the terminal symbol 'comma'.

A *string* is a finite sequence of symbols. We will represent strings by Greek symbols α, β, γ, etc. Thus α = axy is a string over Σ. The length of a string is the number of symbols in it. Note that the absence of any symbol is also a string, the *null string* ε. The *concatenation* operation combines two strings into a single string. It is used to build larger strings from existing strings. Thus, given two strings α and β, concatenation of α with β yields a string which is formed by putting the sequence of symbols forming α before the sequence of symbols forming β. For example, if α = ab, β = axy, then concatenation of α and β, represented as $\alpha.\beta$ or simply $\alpha\beta$, gives the string abaxy. The null string can also participate in a concatenation, thus $a.\varepsilon = \varepsilon.a = a$.

Nonterminal symbols

A *nonterminal symbol* (NT) is the name of a syntax category of a language, e.g. noun, verb, etc. An NT is written as a single capital letter, or as a name enclosed between < ... >, e.g. A or < *Noun* >. During grammatical analysis, a nonterminal symbol represents an instance of the category. Thus, < *Noun* > represents a noun.

Productions

A *production*, also called a *rewriting rule*, is a rule of the grammar. A production has the form

A nonterminal symbol ::= String of Ts and NTs

and defines the fact that the NT on the LHS of the production can be rewritten as the string of Ts and NTs appearing on the RHS. When an NT can be written as one of many different strings, the symbol '|' (standing for 'or') is used to separate the strings on the RHS, e.g.

$$< Article > \quad ::= \quad \texttt{a} \mid \texttt{an} \mid \texttt{the}$$

The string on the RHS of a production can be a concatenation of component strings, e.g. the production

$$< Noun\ Phrase > \quad ::= \quad < Article >< Noun >$$

expresses the fact that the noun phrase consists of an article followed by a noun.

Each grammar G defines a language L_G. G contains an NT called the *distinguished symbol* or the *start NT* of G. Unless otherwise specified, we use the symbol S as the distinguished symbol of G. A valid string α of L_G is obtained by using the following procedure

1. Let α = 'S'.
2. While α is not a string of terminal symbols
   (a) Select an NT appearing in α, say X.
   (b) Replace X by a string appearing on the RHS of a production of X.

Example 1.13 Grammar (1.1) defines a language consisting of noun phrases in English

$$\begin{aligned} < Noun\ Phrase > \quad &::= \quad < Article >< Noun > \\ < Article > \quad &::= \quad \texttt{a} \mid \texttt{an} \mid \texttt{the} \\ < Noun > \quad &::= \quad \texttt{boy} \mid \texttt{apple} \end{aligned} \tag{1.1}$$

< *Noun Phrase* > is the distinguished symbol of the grammar, `the boy` and `an apple` are some valid strings in the language.

Definition 1.6 (Grammar) *A* grammar G *of a language* L_G *is a quadruple* (Σ, *SNT, S, P) where*

Σ *is the alphabet of* L_G*, i.e. the set of* Ts,
SNT *is the set of* NTs,
S *is the distinguished symbol, and*
P *is the set of productions.*

Derivation, reduction and parse trees

A grammar G is used for two purposes, to generate valid strings of L_G and to 'recognize' valid strings of L_G. The derivation operation helps to generate valid strings while the reduction operation helps to recognize valid strings. A parse tree is used to depict the syntactic structure of a valid string as it emerges during a sequence of derivations or reductions.

Derivation

Let production P_1 of grammar G be of the form

$$P_1 : \quad A ::= \alpha$$

and let β be a string such that $\beta \equiv \gamma A \theta$, then replacement of A by α in string β constitutes a *derivation* according to production P_1. We use the notation $N \Rightarrow \eta$ to denote direct derivation of η from N and $N \overset{*}{\Rightarrow} \eta$ to denote transitive derivation of η (i.e. derivation in zero or more steps) from N, respectively. Thus, $A \Rightarrow \alpha$ only if $A ::= \alpha$ is a production of G and $A \overset{*}{\Rightarrow} \delta$ if $A \Rightarrow \ldots \Rightarrow \delta$. We can use this notation to define a valid string according to a grammar G as follows: δ is a valid string according to G only if $S \overset{*}{\Rightarrow} \delta$, where S is the distinguished symbol of G.

Example 1.14 Derivation of the string `the boy` according to grammar (1.1) can be depicted as

$$\begin{aligned} < \textit{Noun Phrase} > &\Rightarrow < \textit{Article} > < \textit{Noun} > \\ &\Rightarrow \texttt{the} < \textit{Noun} > \\ &\Rightarrow \texttt{the boy} \end{aligned}$$

A string α such that $S \overset{*}{\Rightarrow} \alpha$ is a *sentential form* of L_G. The string α is a *sentence* of L_G if it consists of only Ts.

Example 1.15 Consider the grammar G

$$\begin{aligned} < \textit{Sentence} > &::= < \textit{Noun Phrase} >< \textit{Verb Phrase} > \\ < \textit{Noun Phrase} > &::= < \textit{Article} >< \textit{Noun} > \\ < \textit{Verb Phrase} > &::= < \textit{Verb} >< \textit{Noun Phrase} > \\ < \textit{Article} > &::= \texttt{a} \mid \texttt{an} \mid \texttt{the} \\ < \textit{Noun} > &::= \texttt{boy} \mid \texttt{apple} \\ < \textit{Verb} > &::= \texttt{ate} \end{aligned} \tag{1.2}$$

The following strings are sentential forms of L_G.

< *Noun Phrase* >< *Verb Phrase* >
`the boy` < *Verb Phrase* >
< *Noun Phrase* > `ate` < *Noun Phrase* >
`the boy ate` < *Noun Phrase* >
`the boy ate an apple`

However, only `the boy ate an apple` is a sentence.

Reduction

Let production P_1 of grammar G be of the form

$$P_1 : \quad A ::= \alpha$$

and let σ be a string such that $\sigma \equiv \gamma\alpha\theta$, then replacement of α by A in string σ constitutes a *reduction* according to production P_1. We use the notations $\eta \rightarrow N$ and $\eta \xrightarrow{*} N$ to depict direct and transitive reduction, respectively. Thus, $\alpha \rightarrow A$ only if $A ::= \alpha$ is a production of G and $\alpha \xrightarrow{*} A$ if $\alpha \rightarrow \ldots \rightarrow A$. We define the validity of some string δ according to grammar G as follows: δ is a valid string of L_G if $\delta \xrightarrow{*} S$, where S is the distinguished symbol of G.

Example 1.16 To determine the validity of the string

`the boy ate an apple`

according to grammar (1.2) we perform the following reductions

Step	*String*
0	`the boy ate an apple`
1	< *Article* > `boy ate an apple`
2	< *Article* > < *Noun* > `ate an apple`
3	< *Article* > < *Noun* > < *Verb* > `an apple`
4	< *Article* > < *Noun* > < *Verb* > < *Article* > `apple`
5	< *Article* > < *Noun* > < *Verb* > < *Article* > < *Noun* >
6	< *Noun Phrase* > < *Verb* > < *Article* > < *Noun* >
7	< *Noun Phrase* > < *Verb* > < *Noun Phrase* >
8	< *Noun Phrase* > < *Verb Phrase* >
9	< *Sentence* >

The string is a sentence of L_G since we are able to construct the reduction sequence

`the boy ate an apple` $\xrightarrow{*}$ < *Sentence* >.

Parse trees

A sequence of derivations or reductions reveals the syntactic structure of a string with respect to G. We depict the syntactic structure in the form of a *parse tree*. Derivation according to the production $A ::= \alpha$ gives rise to the following elemental parse tree:

(Sequence of Ts and NTs constituting α)

A subsequent step in the derivation replaces an NT in α, say NT$_i$, by a string. We can build another elemental parse tree to depict this derivation, viz.

We can combine the two trees by replacing the node of NT$_i$ in the first tree by this tree. In essence, the parse tree has grown in the downward direction due to a derivation. We can obtain a parse tree from a sequence of reductions by performing the converse actions. Such a tree would grow in the upward direction.

Example 1.17 Figure 1.19 shows the parse tree of the string `the boy ate an apple` obtained using the reductions of Ex. 1.16. The superscript associated with a node in the tree indicates the step in the reduction sequence which led to the subtree rooted at that node. Reduction steps 1 and 2 lead to reduction of `the` and `boy` to <Article> and <Noun>, respectively. Step 3 combines the parse trees of $< Article >$ and $< noun >$ to give the subtree rooted at $< Noun\ Phrase >$.

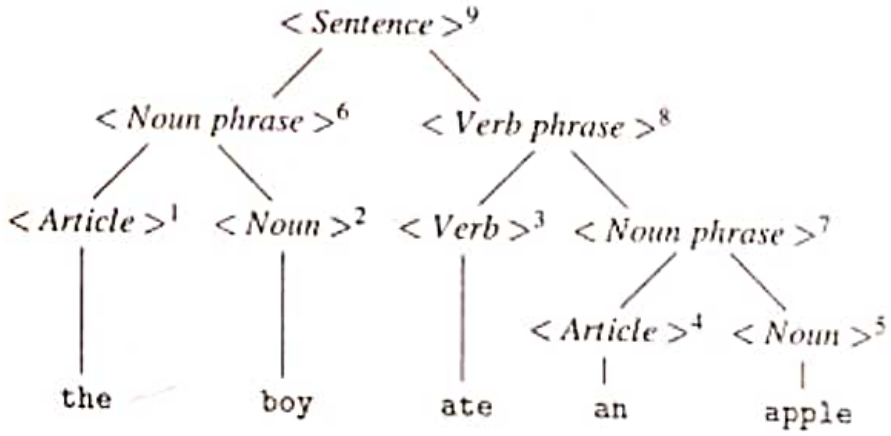


Fig. 1.19 Parse tree

Note that an identical tree would have been obtained if `the boy ate an apple` was derived from S.

Recursive specification

Grammar (1.3) is a complete grammar for an arithmetic expression containing the operators $\uparrow$ (exponentiation), * and +.

$$< exp > \ ::= \ < exp > + < term > | < term >$$

$$
\begin{array}{rcl}
< term > & ::= & < term > * < factor > | < factor > \\
< factor > & ::= & < factor > \uparrow < primary > | < primary > \\
< primary > & ::= & < id > | < constant > | (< exp >) \\
< id > & ::= & < letter > | < id > [< letter > | < digit >] \\
< const > & ::= & [+ | -] < digit > | < const > < digit > \\
< letter > & ::= & \mathtt{a} | \mathtt{b} | \mathtt{c} | \ldots | \mathtt{z} \\
< digit > & ::= & \mathtt{0} | \mathtt{1} | \mathtt{2} | \mathtt{3} | \mathtt{4} | \mathtt{5} | \mathtt{6} | \mathtt{7} | \mathtt{8} | \mathtt{9}
\end{array}
\tag{1.3}
$$

This grammar uses the notation known as the Backus Naur Form (BNF). Apart from the familiar elements ::= , | and < ... >, a new element here is [...], which is used to enclose an optional specification. Thus, the rules for $< id >$ and $< const >$ in grammar (1.3) are equivalent to the rules

$$
\begin{array}{rcl}
< id > & ::= & < letter > | < id > < letter > | < id > < digit > \\
< const > & ::= & < digit > + < digit > | - < digit > \\
 & & | < const > < digit >
\end{array}
$$

Grammar (1.3) uses recursive specification, whereby the NT being defined in a production itself occurs in a RHS string of the production, e.g. X ::= ... X .... The RHS alternative employing recursion is called a *recursive rule*. Recursive rules simplify the specification of recurring constructs.

Example 1.18 A non-recursive specification for expressions containing the '+' operator would have to be written as

$$
\begin{array}{rcl}
< exp > & ::= & < term > | < term > + < term > \\
 & & | < term > + < term > + < term > | \ldots
\end{array}
$$

Using recursion, <*exp*> can be specified simply as

$$
< exp > \quad ::= \quad < exp > + < term > | < term > \tag{1.4}
$$

The first alternative on the RHS of grammar (1.4) is recursive. It permits an unbounded number of '+' operators in an expression. The second alternative is non-recursive. It provides an 'escape' from recursion while deriving or recognizing expressions according to the grammar. Recursive rules are classified into *left-recursive rules* and *right-recursive rules* depending on whether the NT being defined appears on the extreme left or extreme right in the recursive rule. For example, all recursive rules of grammar (1.3) are left-recursive rules. Indirect recursion occurs when two or more NTs are defined in terms of one another. Such recursion is useful for specifying nested constructs in a language. In grammar (1.3), the alternative <*primary*> ::= (<*exp*>) gives rise to indirect recursion because <*exp*> $\overset{+}{\Rightarrow}$ <*primary*>. This

specification permits a parenthesized expression to occur in any context where an identifier or constant can occur.

Direct recursion is not useful in situations where a limited number of occurrences is required. For example, the recursive specification

$$< id > ::= < letter > | < id > [< letter > | < digit >]$$

permits an identifier string to contain an unbounded number of characters, which is not correct. In such cases, controlled recurrence may be specified as

$$< id > ::= < letter > \{< letter > | < digit >\}_0^{15}$$

where the notation $\{\ldots\}_0^{15}$ indicates 0 to 15 occurrences of the enclosed specification.

1.4.1.1 Classification of Grammars

Grammars are classified on the basis of the nature of productions used in them (Chomsky, 1963). Each grammar class has its own characteristics and limitations.

Type-0 grammars

These grammars, known as *phrase structure grammars*, contain productions of the form

$\alpha ::= \beta$

$$\alpha ::= \beta$$

where both α and β can be strings of Ts and NTs. Such productions permit arbitrary substitution of strings during derivation or reduction, hence they are not relevant to specification of programming languages.

$\alpha ::= \beta$

Type-1 grammars

These grammars are known as *context sensitive grammars* because their productions specify that derivation or reduction of strings can take place only in specific contexts. A Type-1 production has the form

$$\alpha\ A\ \beta ::= \alpha\ \pi\ \beta$$

Thus, a string π in a sentential form can be replaced by 'A' (or vice versa) only when it is enclosed by the strings α and β. These grammars are also not particularly relevant for PL specification since recognition of PL constructs is not context sensitive in nature.

Type-2 grammars

These grammars impose no context requirements on derivations or reductions. A typical Type-2 production is of the form

$$A ::= \pi$$

which can be applied independent of its context. These grammars are therefore known as *context free grammars* (CFG). CFGs are ideally suited for programming language specification. Two best known uses of Type-2 grammars in PL specification are the ALGOL-60 specification (Naur, 1963) and Pascal specification (Jensen, Wirth, 1975). The reader can verify that grammars (1.2) and (1.3) are Type-2 grammars.

Type-3 grammars

Type-3 grammars are characterized by productions of the form

$$\begin{aligned} A &::= \mathtt{t}B \mid \mathtt{t} \text{ or} \\ A &::= B\mathtt{t} \mid \mathtt{t} \end{aligned}$$

Note that these productions also satisfy the requirements of Type-2 grammars. The specific form of the RHS alternatives—namely a single T or a string containing a single T and a single NT—gives some practical advantages in scanning (we shall see this aspect in Chapter 6). However, the nature of the productions restricts the expressive power of these grammars, e.g. nesting of constructs or matching of parentheses cannot be specified using such productions. Hence the use of Type-3 productions is restricted to the specification of lexical units, e.g. identifiers, constants, labels, etc. The productions for <*constant*> and <*identifier*> in grammar (1.3) are in fact Type-3 in nature. This can be seen clearly when we rewrite the production for $< id >$ in the form B t | t, viz.

$$< id > ::= l \mid < id > l \mid < id > d$$

where l and d stand for a letter and digit respectively.

Type-3 grammars are also known as *linear grammars* or *regular grammars*. These are further categorized into left-linear and right-linear grammars depending on whether the NT in the RHS alternative appears at the extreme left or extreme right.

Operator grammars

Definition 1.7 (Operator grammar (OG)) *An* operator grammar *is a grammar none of whose productions contain two or more consecutive NTs in any RHS alternative.*

Thus, nonterminals occurring in an RHS string are separated by one or more terminal symbols. All terminal symbols occurring in the RHS strings are called

operators of the grammar. As we will discuss later in Chapter 6, OGs have certain practical advantages in compiler writing.

Example 1.19 Grammar (1.3) is an OG. '↑', '*', '+', '(' and ')' are the operators of the grammar.

1.4.1.2 Ambiguity in Grammatic Specification

Ambiguity implies the possibility of different interpretations of a source string. In natural languages, ambiguity may concern the meaning or syntax category of a word, or the syntactic structure of a construct. For example, a word can have multiple meanings or can be both noun and verb (e.g. the word 'base'), and a sentence can have multiple syntactic structures (e.g. 'police ordered to stop speeding on roads'). Formal language grammars avoid ambiguity at the level of a lexical unit or a syntax category. This is achieved by the simple rule that identical strings cannot appear on the RHS of more than one production in the grammar. Existence of ambiguity at the level of the syntactic structure of a string would mean that more than one parse tree can be built for the string. In turn, this would mean that the string can have more than one meaning associated with it.

Example 1.20 Consider the expression grammar

$$\begin{aligned} <exp> &::= <id> \mid <exp> + <exp> \mid <exp> * <exp> \\ <id> &::= \texttt{a} \mid \texttt{b} \mid \texttt{c} \end{aligned} \tag{1.5}$$

Two parse trees exist for the source string a+b*c according to this grammar —one in which a+b is first reduced to <*exp*> and another in which b*c is first reduced to <*exp*>. Since semantic analysis derives the meaning of a string on the basis of its parse tree, clearly two different meanings can be associated with the string.

Eliminating ambiguity

An ambiguous grammar should be rewritten to eliminate ambiguity. In Ex. 1.20, the first tree does not reflect the conventional meaning associated with a+b*c, while the second tree does. Hence the grammar must be rewritten such that reduction of '*' precedes the reduction of '+' in a+b*c. The normal method of achieving this is to use a hierarchy of NTs in the grammar, and to associate the reduction or derivation of an operator with an appropriate NT.

Example 1.21 Figure 1.20 illustrates reduction of a+b*c according to Grammar 1.3. Part (a) depicts an attempt to reduce a+b to <*exp*>. This attempt fails because the resulting string <*exp*> * <*id*> cannot be reduced to <*exp*>. Part (b) depicts the correct reduction of a+b*c in which b*c is first reduced to <*exp*>. This sequence of reductions can be explained as follows: Grammar (1.3) associates the recognition of '*' with reduction of a string to a <*term*>, which alone can take part in a reduction involving '+'. Consequently, in a+b*c, '*' has to be necessarily reduced before '+'. This yields the conventional meaning of the string. Other NTs, viz. <*factor*> and <*primary*>, similarly take care of the operator '↑' and the parentheses '(...)'. Hence there is no ambiguity in grammar (1.3).

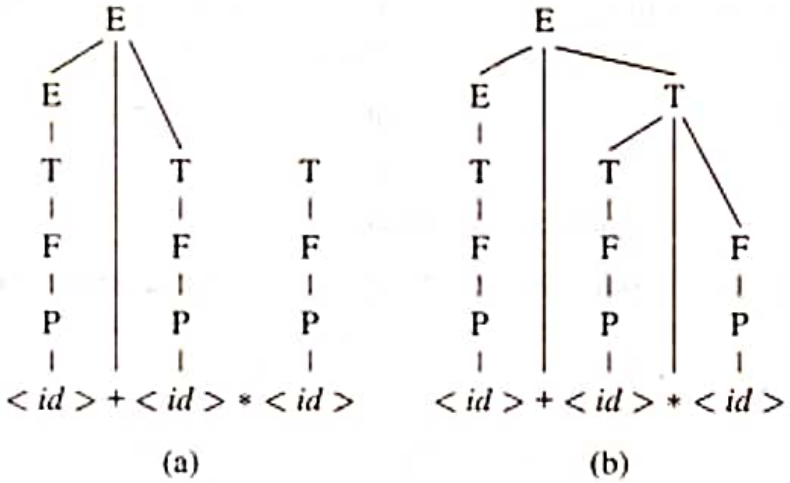


Fig. 1.20 Ensuring a unique parse tree for an expression

EXERCISE 1.4

1. In grammar (1.3), identify productions which could belong to
   (a) an operator grammar.
   (b) a linear grammar.
2. Write productions for the following
   (a) a decimal constant with or without a fractional part,
   (b) a real number with mantissa and exponent specification.
3. In grammar (1.3) what are the priorities of '+', '*' and '↑' with respect to one another?
4. In grammar (1.3) add productions to incorporate relational and boolean operators.
5. *Associativity* of an operator indicates the order in which consecutive occurrences of the operator in a string are reduced. For example '+' is left associative, i.e. in `a+b+c`, `a+b` is performed first, followed by the addition of `c` to its result.
   (a) Find the associativities of operators in grammar (1.3).
   (b) Exponentiation should be right associative so that `a↑b↑c` has the conventional meaning a^{b^c}. What changes should be made in grammar (1.3) to implement right associativity for ↑?
   (c) Is the grammar of problem 3 of Exercise 3.2.2 ambiguous? If so, give a string which has multiple parses.

1.4.2 Binding and Binding Times

Each program entity pe_i in program P has a set of attributes $A_i \equiv \{a_j\}$ associated with it. If pe_i is an identifier, it has an attribute *kind* whose value indicates whether it is a variable, a procedure or a reserved identifier (i.e. a keyword). A variable has attributes like type, dimensionality, scope, memory address, etc. Note that the attribute of one program entity may be another program entity. For example, type is

an attribute of a variable. It is also a program entity with its own attributes, e.g. size (i.e. number of memory bytes). The values of the attributes of a type `typ` should be determined some time before a language processor processes a declaration statement using that type, viz. a Pascal-like statement

```
var : type;
```

For simplicity we often use the words 'the a_j attribute of pe_i is ..' instead of the more precise words 'the value of the a_j attribute of pe_i is ..'.

Definition 1.8 (Binding) *A* binding *is the association of an attribute of a program entity with a value.*

Binding time is the time at which a binding is performed. Thus the type attribute of variable `var` is bound to `typ` when its declaration is processed. The size attribute of `typ` is bound to a value sometime prior to this binding. We are interested in the following binding times:

1. Language definition time of L
2. Language implementation time of L
3. Compilation time of P
4. Execution init time of `proc`
5. Execution time of `proc`.

where L is a programming language, P is a program written in L and `proc` is a procedure in P. Note that language implementation time is the time when a language translator is designed. The preceding list of binding times is not exhaustive; other binding times can be defined, viz. binding at the linking time of P. The language definition of L specifies binding times for the attributes of various entities of a program written in L.

Example 1.22 Consider the Pascal program

```
program bindings(input, output);
   var
      i : integer;
      a,b : real;
   procedure proc (x : real; j : integer);
      var
         info : array [1..10, 1..5] of integer;
         p : ↑integer;
      begin
         new(p);
   end;
   begin
   proc(a,i);
end.
```

Binding of the keywords of Pascal to their meanings is performed at language definition time. This is how keywords like **program**, **procedure**, **begin** and **end** get their meanings. These bindings apply to all programs written in Pascal. At language implementation time, the compiler designer performs certain bindings. For example, the size of type 'integer' is bound to n bytes where n is a number determined by the architecture of the target machine. Binding of type attributes of variables is performed at compilation time of program `bindings`. The memory addresses of local variables `info` and `p` of procedure `proc` are bound at every execution init time of procedure `proc`. The value attributes of variables are bound (possibly more than once) during an execution of `proc`. The memory address of `p↑` is bound when the procedure call `new (p)` is executed.

Importance of binding times

The binding time of an attribute of a program entity determines the manner in which a language processor can handle the use of the entity. A compiler can generate code specifically tailored to a binding performed during or before compilation time. However, a compiler cannot generate such code for bindings performed later than compilation time. This affects execution efficiency of the target program.

Example 1.23 Consider the PL/1 program segment

```
procedure pl1_proc (x, j, info_size, columns);
   declare x float;
   declare (j, info_size, columns) fixed;
   declare pl1_info (1:info_size,1:columns) fixed;
   ...
end pl1_proc;
```

Here the size of array `pl1_info` is determined by the values of parameters `info_size` and `columns` in a specific call of `pl1_proc`. This is an instance of execution time binding. The compiler does not know the size of array `pl1_info`. Hence it may not be able to generate efficient code for accessing its elements.

The dimension bounds of array `info` in program `bindings` of Ex. 1.22 are constants. Thus, binding of the dimension bound attributes can be performed at compilation time. This enables the Pascal compiler to generate efficient code to access elements of `info`. Thus the PL/1 program of Ex. 1.23 may execute slower than the Pascal program of Ex. 1.22 (see Section 6.2.3 for more details). However, the PL/1 program provides greater flexibility to the programmer since the dimension bounds can be specified during program execution. From this comparison, we can draw the following inference concerning the influence of binding times on the characteristics of programs: An early binding provides greater execution efficiency whereas a late binding provides greater flexibility in the writing of a program.

Static and dynamic bindings

Definition 1.9 (Static binding) A static binding *is a binding performed before the execution of a program begins.*

Definition 1.10 (Dynamic binding) *A* dynamic binding *is a binding performed after the execution of a program has begun.*

Needless to say that static bindings lead to more efficient execution of a program than dynamic bindings. We shall discuss static and dynamic binding of memory in Section 6.2.1 and dynamic binding of variables to types in Section 6.6.

1.5 LANGUAGE PROCESSOR DEVELOPMENT TOOLS

The analysis phase of a language processor has a standard form irrespective of its purpose, the source text is subjected to lexical, syntax and semantic analysis and the results of analysis are represented in an IR. Thus writing of language processors is a well understood and repetitive process which ideally suits the program generation approach to software development. This has led to the development of a set of language processor development tools (LPDTs) focussing on generation of the analysis phase of language processors.

Figure 1.21 shows a schematic of an LPDT which generates the analysis phase of a language processor whose source language is L. The LPDT requires the following two inputs:

1. Specification of a grammar of language L
2. Specification of semantic actions to be performed in the analysis phase.

It generates programs that perform lexical, syntax and semantic analysis of the source program and construct the IR. These programs collectively form the analysis phase of the language processor (see the dashed box in Fig. 1.21).

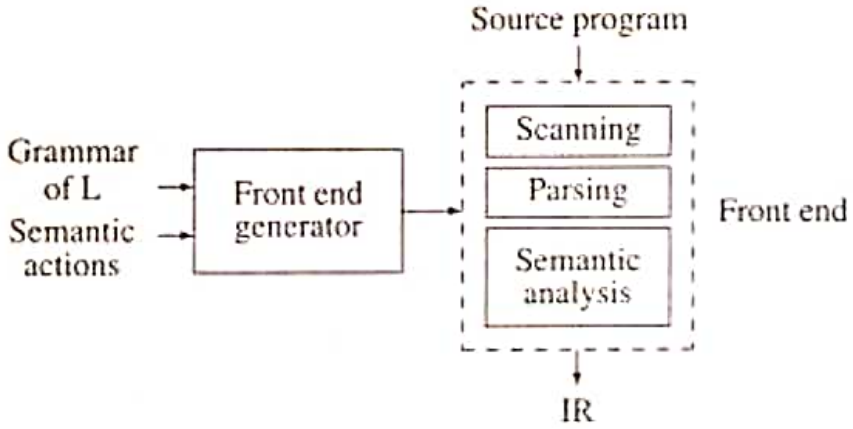


Fig. 1.21 A language processor development tool (LPDT)

We briefly discuss two LPDTs widely used in practice. These are, the lexical analyser generator LEX, and the parser generator YACC. The input to these tools is a specification of the lexical and syntactic constructs of L, and the semantic actions to be performed on recognizing the constructs. The specification consists of a set of *translation rules* of the form

$$< string\ specification > \quad \{ < semantic\ action > \}$$

where < *semantic action* > consists of C code. This code is executed when a string matching < *string specification* > is encountered in the input. LEX and YACC generate C programs which contain the code for scanning and parsing, respectively, and the semantic actions contained in the specification. A YACC generated parser can use a LEX generated scanner as a routine if the scanner and parser use same conventions concerning the representation of tokens. Figure 1.22 shows a schematic for developing the analysis phase of a compiler for language L using LEX and YACC. The analysis phase processes the source program to build an intermediate representation (IR). A single pass compiler can be built using LEX and YACC if the semantic actions are aimed at generating target code instead of IR. Note that the scanner also generates an intermediate representation of a source program for use by the parser. We call it IR_l in Fig. 1.22 to differentiate it from the IR of the analysis phase.

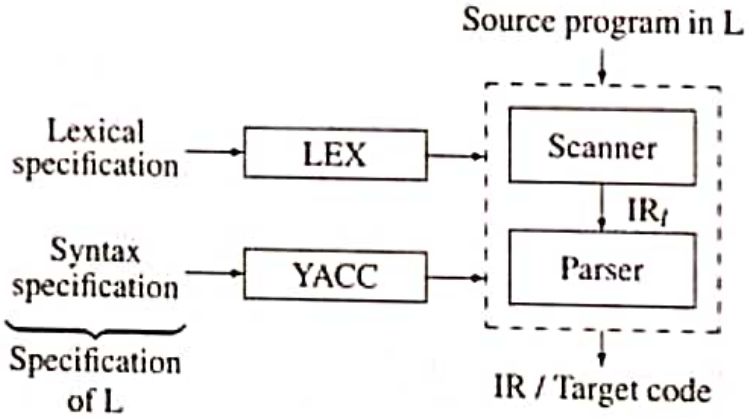


Fig. 1.22 Using LEX and YACC

1.5.1 LEX

LEX accepts an input specification which consists of two components. The first component is a specification of strings representing the lexical units in L, e.g. id's and constants. This specification is in the form of regular expressions defined in Section 3.1. The second component is a specification of semantic actions aimed at building an IR. As seen in Section 1.3.1.1, the IR consists of a set of tables of lexical units and a sequence of tokens for the lexical units occurring in a source statement. Accordingly, the semantic actions make new entries in the tables and build tokens for the lexical units.

Example 1.24 Figure 1.23 shows a sample input to LEX. The input consists of four components, three of which are shown here. The first component (enclosed by %{ and %}) defines the symbols used in specifying the strings of L. It defines the symbol `letter` to stand for any upper or lower case letter, and `digit` to stand for any digit. The second component enclosed between %% and %% contains the translation rules. The third component contains auxiliary routines which can be used in the semantic actions.

```
%{
letter                          [A-Za-z]
digit                           [0-9]
}%

%%
begin                           {return(BEGIN);}
end                             {return(END);}
":= "                           {return(ASGOP);}
{letter} ({letter}|{digit})*    {yylval=enter_id();
                                 return(ID);}
{digit}+                        {yylval=enter_num();
                                 return(NUM);}
%%
enter_id()
{ /* enters the id in the symbol table and returns
     entry number */ }
enter_num()
{ /* enters the number in the constants table and
     returns entry number */}
```

Fig. 1.23 A sample LEX specification

The sample input in Figure 1.23 defines the strings `begin`, `end`, `:=` (the assignment operator), and identifier and constant strings of L. When an identifier is found, it is entered in the symbol table (if not already present) using the routine `enter_id`. The pair (`ID`, *entry #*) forms the token for the identifier string. By convention *entry #* is put in the global variable `yylval`, and the class code `ID` is returned as the value of the call on scanner. Similar actions are taken on finding a constant, the keywords `begin` and `end` and the assignment operator. Note that each operator and keyword has been made into a class by itself to suit the conventions mentioned earlier.

1.5.2 YACC

Each string specification in the input to YACC resembles a grammar production. The parser generated by YACC performs reductions according to this grammar. The actions associated with a string specification are executed when a reduction is made according to the specification. An attribute is associated with every nonterminal symbol. The value of this attribute can be manipulated during parsing. The attribute can be given any user-designed structure. A symbol '$*n*' in the action part of a translation rule refers to the attribute of the nth symbol in the RHS of the string specification. '$$' represents the attribute of the LHS symbol of the string specification.

Example 1.25 Figure 1.24 shows sample input to YACC. The input consists of four components, of which only two are shown. It is assumed that the attribute of a symbol resembles the attributes used in Fig. 1.15. The routine `gendesc` builds a descriptor containing the name and type of an id or constant. The routine `gencode` takes an operator and the attributes of two operands, generates code and returns with the attribute

for the result of the operation. This attribute is assigned as the attribute of the LHS symbol of the string specification. In a subsequent reduction this attribute becomes the attribute of some RHS symbol.

```
%%
E : E+T   {$$ = gencode('+', $1, $3);}
  | T     {$$ = $1;}
  ;
T : T*V   {$$ = gencode('*', $1, $3);}
  | V     {$$ = $1;}
  ;
V : id    {$$ = gendesc($1);}
  ;
%%
gencode (operator, operand_1, operand_2)
{ /* Generates code using operand descriptors.
     Returns descriptor for result */ }
gendesc (symbol)
{ /* Refer to symbol/constant table entry.
     Build and return descriptor for the symbol */ }
```

Fig. 1.24 A sample YACC specification

Parsing of the string b+c*d where b, c and d are of type real, using the parser generated by YACC from the input of Fig. 1.24 leads to following calls on the C routines:

```
Gendesc ( Id #1 );
Gendesc ( Id #2 );
Gendesc ( Id #3 );
Gencode ( *, c, real, d, real );
Gencode ( +, b, real, t, real );
```

where an attribute has the form <name>, <type>, and t is the name of a location (a register or memory word) used to store the result of c*d in the code generated by the first call on gencode. Details of gencode shall be discussed in Chapter 6.

BIBLIOGRAPHY

Elson (1973), Tennent (1981), Pratt (1983) and Maclennan (1983) discuss the influence of programming language design on aspects of compilation and execution, i.e. on the specification and execution gaps. Cleaveland and Uzgalis (1977) and Backhouse (1979) are devoted to programming language grammars. Programming language grammars are also discussed in most compiler books.

Prywes *et al* (1979) discusses automatic program generation. Martin (1982) describes generation of application programs.

(a) Programming languages

1. Elson, M. (1973): *Concepts of Programming Languages*, Science Research Associates, Chicago.

2. Maclennan, B.J. (1983): *Principles of Programming Languages*, Holt, Rinehart & Winston, New York.
3. Pratt, T.W. (1983): *Programming Languages – Design and Implementation*, Prentice-Hall, Englewood Cliffs.
4. Tennent, R.D. (1981): *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs.

(b) Grammars for programming languages

1. Backhouse, R.C. (1979): *Syntax of Programming Languages*, Prentice-Hall, New Jersey.
2. Cleaveland, J.C. and R.C. Uzgalis (1977): *Grammars for Programming Languages*, Elsevier, New York.
3. Jensen, K. and N. Wirth (1975): *Pascal User Manual and Report*, Springer-Verlag, New York.
4. Lewis, P.M., D.J. Rosenkrantz and R.E. Stearns (1976): *Compiler Design Theory*, Addison-Wesley, Reading.
5. Naur, P. (ed.) (1963): "Revised report on the algorithmic language ALGOL 60," *Commn. of ACM*, **6** (1), 1.
6. Naur, P. (1975): "Programming languages, natural languages and mathematics, *Commn. of ACM*, **18** (12), 676-683.

(c) Program generation

1. Martin, J. (1982): *Application Development Without Programmers*, Prentice-Hall, Englewood Cliffs.
2. Prywes, N.S., A. Pnueli and S. Shastry (1979): "Use of nonprocedural specification language and associated program generator in software development," *ACM TOPLAS*, **1** (2), 196-217.

(d) Language processor development tools

1. Graham, S.L. (1980): "Table driven code generation," *Computer*, **13** (8), 25-34.
2. Johnson, S.C. (1975): "Yacc – Yet Another Compiler Compiler," *Computing Science Technical Report no. 32*, AT & T Bell Laboratories, N.J.
3. Lesk, M.E. (1975): "Lex – A lexical analyzer generator," *Computing Science Technical Report no. 39*, At & T Bell Laboratories, N.J.
4. Schreiner, A.T. and H.G. Fredman, Jr. (1985): *Introduction to Compiler Construction with Unix*, Prentice-Hall, Englewood Cliffs.
5. Levine, J.R., T. Mason and D. Brown (1990): *Lex and Yacc*, 2nd edition, O'Reilly & associates, Sebastopol.

CHAPTER 2

Data Structures for Language Processing

The space–time tradeoff in data structures, i.e. the tradeoff between memory requirements and the search efficiency of a data structure, is a fundamental principle of systems programming. A language processor makes frequent use of the search operation over its data structures. This makes the design of data structures a crucial issue in language processing activities. In this chapter we shall discuss the data structure requirements of language processors and suggest efficient data structures to meet these requirements.

The data structures used in language processing can be classified on the basis of the following criteria:

1. Nature of a data structure—whether a *linear* or *nonlinear* data structure
2. Purpose of a data structure—whether a *search* data structure or an *allocation* data structure.
3. Lifetime of a data structure—whether used during language processing or during target program execution.

A *linear data structure* consists of a linear arrangement of elements in the memory. The physical proximity of its elements is used to facilitate efficient search. However, a linear data structure requires a contiguous area of memory for its elements. This poses problems in situations where the size of a data structure is difficult to predict. In such a situation, a designer is forced to overestimate the memory requirements of a linear data structure to ensure that it does not outgrow the allocated memory. This leads to wastage of memory. The elements of a *nonlinear data structure* are accessed using pointers. Hence the elements need not occupy contiguous areas of memory, which avoids the memory allocation problem seen in the context

of linear data structures. However, the nonlinear arrangement of elements leads to lower search efficiency.

Example 2.1 Figure 2.1(a) shows memory allocation for four linear data structures. Parts of the data structures shaded with dotted lines are not in current use. Note that these parts may remain unused throughout the execution of the program, however the memory allocated to them cannot be used for other data structure(s). Figure 2.1(b) shows allocation to four nonlinear data structures. Elements of the data structures are allocated noncontiguous areas of memory as and when needed. This is how two memory areas are allocated to E while three memory areas are allocated to F and two memory areas are allocated to H. No parts of the data structures are currently unused. Free memory existing in the system can be allocated to any new or existing data structures.

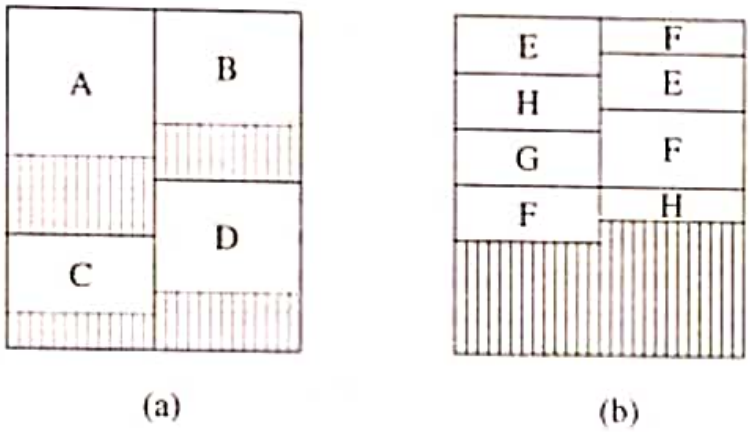


Fig. 2.1 Linear and nonlinear data structures

Search data structures are used during language processing to maintain attribute information concerning different entities in the source program. These data structures are characterized by the fact that the entry for an entity is created only once but may be searched for a large number of times. Search efficiency is therefore very important. *Allocation data structures* are characterized by the fact that the address of the memory area allocated to an entity is known to the user(s) of that entity. Thus no search operations are conducted on them. Speed of allocation or deallocation and efficiency of memory utilization are the important criteria for the allocation data structures.

A language processor uses both search and allocation data structures during its operation (see Ex. 2.2). Search data structures are used to constitute various tables of information. Allocation data structures are used to handle programs with nested structures of some kind. A target program rarely uses search data structures. However, it may use allocation data structures (see Ex. 2.3).

Example 2.2 Consider the Pascal program

```
Program Sample(input,output);
  var
      x,y : real;
      i   : integer;
  Procedure calc(var a,b : real);
    var
      sum : real;
    begin
      sum := a+b;
      ...
  end calc;
  begin { Main program }
  ...
end.
```

The definition of procedure `calc` is nested inside the main program. Symbol tables need to be created for the main program as well as for procedure `calc`. Let us call these $Symtab_{Sample}$ and $Symtab_{calc}$, respectively. These tables are search data structures for obvious reasons. During compilation, the attributes of a symbol, e.g. symbol a, are obtained by searching the appropriate symbol table. Memory needs to be allocated to $Symtab_{Sample}$ and $Symtab_{calc}$ using an allocation data structure. The addresses of these tables are noted in a suitable manner. Hence no searches are involved in locating $Symtab_{Sample}$ or $Symtab_{calc}$.

Example 2.3 Consider the following Pascal and C program segments

```
Pascal  :  var p : ↑integer;
           begin
             new(p);

C       :  float *ptr;
           ptr = (float *) calloc(5, sizeof(float));
```

The Pascal call `new(p)` allocates sufficient memory to hold an integer value and puts the address of this memory area in p. The C statement `ptr = ..` allocates a memory area sufficient to hold 5 float values and puts its address in `ptr`. Accesses to these memory areas are implemented through pointers p and `ptr`. No search is involved in accessing the allocated memory.

2.1 SEARCH DATA STRUCTURES

A search data structure (or *search structure* for short) is a set of entries, each entry accommodating the information concerning one entity. Each entry is assumed to contain a *key* field which forms the basis for a search. Throughout this section we will use examples of entries in a symbol table used by a language processor. Here, the key field is the *symbol* field containing the name of an entity.

Entry formats

Each entry in a search structure is a set of fields. It is common for an entry in a search structure to consist of two parts, a fixed part and a variant part. Each part consists of a set of fields. Fields of the fixed part exist in each entry of the search structure. The value in a *tag* field of the fixed part determines the information to be stored in the variant part of the entry. For each value v_i in the tag field, the variant part of the entry consists of the set of fields SF_{v_i} (see Ex. 2.4). Note that the fixed and variant parts may be nested, i.e. a variant part may itself consist of fixed and variant parts, etc.

Example 2.4 Entries in the symbol table of a compiler have the following fields:

Fixed part: Fields *symbol* and *class* (*class* is the tag field).

Variant part:

Tag value	*Variant part fields*
variable	*type, length, dimension info.*
procedure name	*address of parameter list, number of parameters.*
function name	*type of returned value, length of returned value, address of parameter list, number of parameters.*
label	*statement number.*

Fixed and variable length entries

An entry may be declared as a *record* or a *structure* of the language in which the language processor is being implemented. (We shall use the word 'record' in our discussion.) In the *fixed length entry* format, each record is defined to consist of the following fields:

1. Fields in the fixed part of the entry
2. $\bigcup_{v_i} SF_{v_i}$, i.e. the set of fields in all variant parts of the entry.

All records in the search structure now have an identical format. This enables the use of homogeneous linear data structures like arrays. In turn, the use of linear organizations enables the use of efficient search procedures (we shall see this later in the section). However, this organization makes inefficient use of memory since many records may contain redundant fields.

In the *variable length entry* format, a record consists of the following fields:

1. Fields in the fixed part of the entry, including the tag field
2. $\{ f_j \mid f_j \in SF_{v_j} \text{ if } tag = v_j \}$.

This entry format leads to a compact organization in which no memory wastage occurs.

(a)

1	2	3	4	5	6	7	8	9	10

1. symbol
2. class
3. type
4. length
5. dimension information
6. parameter list address
7. no. of parameters
8. type of returned value
9. length of returned value
10. statment number

(b)

1	2	3

1. name 2. class 3. statement number

Fig. 2.2 (a) Fixed entry, (b) Variable length entry for label

Example 2.5 Figure 2.2 shows two entry formats for the symbol table of Ex. 2.4. Part (a) shows the fixed length entry format. When *class* = label, all fields excepting *name*, *class* and *statement number* are redundant. Part (b) shows the variable length entry format when *class* = label.

When a variable length entry format is used, the search method may require knowledge of the length of an entry. In such cases a record would consist of the following fields:

1. A *length* field
2. Fields in the fixed part of the entry, including the tag field
3. $\{ f_j \mid f_j \in SF_{v_j} \text{ if } tag = v_j \}$.

We will depict this format as

length	*entry*

Hybrid entry formats

A Hybrid entry format is used as a compromise between the fixed and variable entry formats to combine the access efficiency of the fixed entry format with the memory efficiency of the variable entry format. In this format each entry is split into two halves, the fixed part and the variable part. A *pointer* field is added to the fixed part. It points to the variable part of the entry. The fixed and variable parts are accommodated in two different data structures. The fixed parts of all entries are organized into an efficient search structure, e.g. a linear data structure. Since the fixed part contains a pointer to the variable part, the variable part does not need to be located through a search. Hence it is put into an allocation data structure which can be linear or nonlinear in nature. The hybrid entry format is depicted as

fixed part	*pointer*

length	*variable part*

Operations on search structures

The following operations are performed on search data structures:

1. Operation *add:* Add the entry of a symbol.
2. Operation *search:* Search and locate the entry of a symbol.
3. Operation *delete:* Delete the entry of a symbol.

The entry for a symbol is created only once, but may be searched for a large number of times during the processing of a program. The deletion operation is not very common.

Generic search procedure

We give the generic procedure to search and locate the entry of symbol s in a search data structure as Algorithm 2.1.

Algorithm 2.1 (Generic search procedure)

1. Make a prediction concerning the entry of the search data structure which symbol s may be occupying. Let this be entry e.
2. Let s_e be the symbol occupying e^{th} entry. Compare s with s_e. Exit with success if the two match.
3. Repeat steps 1 and 2 till it can be concluded that the symbol does not exist in the search data structure.

The nature of the prediction varies with the organization of the search data structure. Each comparison of Step 2 is called a *probe*. Efficiency of a search procedure is determined by the number of probes performed by the search procedure. We use following notation to represent the number of probes in a search:

p_s : Number of probes in a successful search
p_u : Number of probes in an unsuccessful search

2.1.1 Table Organizations

A table is a linear data structure. The entries of a table occupy adjoining areas of the memory. Two points can be made concerning tables as search structures:

1. Given the location of an entry of the table, it is meaningful to talk of the *next* entry of the table or the *previous* entry of the table. A search technique may use this fact to advantage.
2. Tables using the fixed length entry organization possess the property of *positional determinacy*. This property states that the address of an entry in a table can be determined from its entry number. For example, the address of the eth

entry is $a+(e-1).l$, where a is the address of the first entry and l is the length of an entry. This property facilitates the representation of a symbol s by e, its entry number in the search structure, in the intermediate code generated by a language processor (see Section 1.3.1). Positional determinacy may also be used to design efficient search procedures, as we shall see later in this section. Tables using variable length entries do not possess the property of positional determinacy, so one must step through the first $(e-1)$ entries of a table in order to locate the eth entry. Hence variable length entries are generally avoided in linear data structures. In our discussion, we assume the use of fixed length entries in linear data structures.

Sequential search organization

Figure 2.3 shows a typical state of a table using the sequential search organization. We use the following symbols in our discussion:

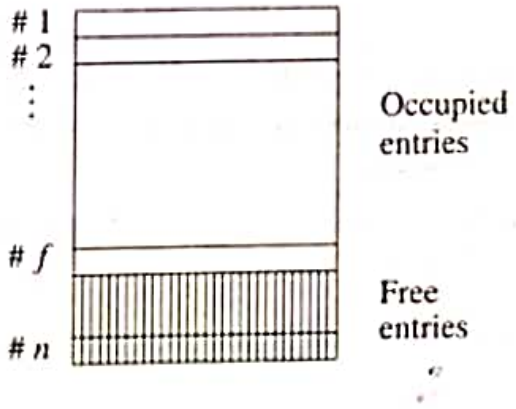


Fig. 2.3 Sequential search table

n : Number of entries in the table
f : Number of occupied entries

Search for a symbol

At any stage the search prediction in Algorithm 2.1 is that symbol s occupies the *next* entry of the table, where *next* = 1 to start with. From Algorithm 2.1, it follows that if all active entries in the table have the same probability of being accessed, we have

$$p_s = f/2 \quad \text{for a successful search}$$
$$p_u = f \quad \text{for an unsuccessful search}$$

Following an unsuccessful search, a symbol may be entered in the table using an *add* operation.

Add a symbol

The symbol is added to the first free entry in the table. The value of f is updated accordingly.

Delete a symbol

Deletion of an entry can be implemented in two ways, physical deletion and logical deletion. In *physical deletion*, an entry is deleted by erasing or by overwriting. Thus, if the dth entry is to be deleted, entries $d+1$ to f can be shifted 'up' by one entry each. This would require $(f-d)$ shift operations in the symbol table. An efficient alternative would be to move the fth entry into the dth position, thus requiring only one shift operation. Physical deletion causes changes in the entry numbers of symbols, which interferes with the representation of a symbol in the IC. Hence physical deletion is seldom used.

Logical deletion of an entry is performed by adding some information to the entry to indicate its deletion. This can be implemented by introducing a field to indicate whether an entry is active or deleted. The complete symbol table entry now looks as follows:

Active/deleted	*Symbol*	*Other info*

Binary search organization

All entries in a table are assumed to satisfy an ordering relation. For example, use of the '<' relation implies that the symbol occupying an entry is 'smaller than' the symbol occupying the next entry. At any stage the search prediction in Algorithm 2.1 is that s occupies the middle entry of that part of the table which is expected to contain its entry.

Algorithm 2.2 (Binary search)

1. $start := 1$; $end := f$;
2. While $start \leq end$
   (a) $e := \lceil \frac{start+end}{2} \rceil$; where $\lceil \ldots \rceil$ implies a rounded quotient. Exit with success if $s = s_e$.
   (b) If $s < s_e$ then $end := e - 1$;
   else $start := e + 1$;
3. Exit with failure.

For a table containing f entries, we have $p_s \leq \lceil log_2 f \rceil$ and $p_u = \lceil log_2 f \rceil$. Thus the search performance is logarithmic in the size of the table. However, the requirement that the entry number of a symbol in the table should not change after an *add* operation (due to its use in the IC), forbids both additions and deletions during language processing. Hence, binary search organization is suitable only for a table

containing a fixed set of symbols, e.g. the table of keywords in a PL. For the same reason, it cannot be used for a symbol table unless one can afford a separate pass of the language processor for constructing the table.

Hash table organization

In the hash table organization the search prediction in Algorithm 2.1 depends on the value of s, i.e. e is a function of s. Three possibilities exist concerning the predicted entry—the entry may be occupied by s, the entry may be occupied by some other symbol, or the entry may be empty. The situation in the second case, i.e. $s \neq s_e$, is called a *collision*. Following a collision, the search continues with a new prediction. In the third case, s is entered in the predicted entry.

Algorithm 2.3 (Hash table management)

1. $e := h(s)$;
2. Exit with success if $s = s_e$, and with failure if entry e is unoccupied.
3. Repeat steps 1 and 2 with different functions h', h'', etc.

The function h used in Algorithm 2.3 is called a *hashing function*. We use the following notation to discuss the properties of hashing functions:

n	:	Number of entries in the table
f	:	Number of occupied entries in the table
ρ	:	*Occupation density* in the table, i.e. f/n
k	:	Number of distinct symbols in the source language
k_p	:	Number of symbols used in some source program
S_p	:	Set of symbols used in some source program
N	:	Address space of the table, i.e. the space formed by the entries $1 \ldots n$
K	:	Key space of the system, i.e. the space formed by enumerating all symbols of the source language. We will denote it as $1 \ldots k$
K_p	:	Key space of a program, i.e. $1 \ldots k_p$

A hashing function has the property that $1 \leq h(symb) \leq n$, where *symb* is any valid symbol of the source language. If $k \leq n$, we can select a one-to-one function as the hashing function h. This will eliminate collisions in the symbol table since entry number e given by $e = h(s)$ can only be occupied by symbol s. We refer to this organization as a *direct entry organization*.

However, k is a very large number in practice hence use of a one-to-one function will require a very large symbol table. Fortunately a one-to-one function is not

needed. For good search performance it is adequate if the hashing function implements a mapping $K_p \Rightarrow N$ which is nearly one-to-one for any arbitrary set of symbols S_p.

The effectiveness of a hashing organization depends on the average value of p_s. For a given size of a hash table, the value of p_s can be expected to increase with the value of k_p.

Hashing functions

While hashing, the representation of s, the symbol to be searched, is treated as a binary number. The hashing function performs a numerical transformation on this number to obtain e. Let the representation of s have b bits in it and let the host computer use m bit arithmetic. Now, to apply the numerical transformation, we need to obtain an m bit representation of s. Let us call it r_s. If $b \leq m$, the representation of s can be padded with 0's to obtain r_s. If $b > m$, the representation of s is split into pieces of m bits each, and bitwise exclusive OR operations are performed on these pieces to obtain r_s. This process is called *folding*. The hashing function h is now applied to r_s. This method ensures that the effect of all characters in a symbol is incorporated during hashing.

A hashing function h should possess the following properties to ensure good search performance:

1. The hashing function should not be sensitive to the symbols in S_p, that is, it should perform equally well for different source programs. Thus, the value of p_s should only depend on k_p.
2. The hashing function h should execute reasonably fast.

The first property is satisfied by designing a hashing function which is a good randomizer over the table space. This makes its performance insensitive to S_p. Two popular classes of such hashing functions are described in the following.

1. *Multiplication functions:* These functions are analogous to functions used in random number generation, e.g. $h(s) = (a \times r_s + b) \bmod 2^m$, where a, b are constants and fixed point arithmetic is used to compute $h(s)$. The table size should be a power of 2, say 2^g, such that the lower order g bits of $h(s)$ can be used as e.
2. *Division functions:* A typical division hashing function is

$$h(s) = (remainder\ of\ \frac{r_s}{n}) + 1$$

where n is the size of the table. If n is a prime number, the method is called *prime division hashing*.

Both multiplication and division methods perform well in practice. A multiplication method has the advantage of being slightly faster but suffers from the drawback

that the table size has to be a power of 2. Prime division hashing is slower but has the advantage that prime numbers are much more closely spaced than powers of 2. This provides a wider choice of table size.

Collision handling methods

Two approaches to collision handling are to accommodate a colliding entry elsewhere in the hash table using a *rehashing* technique, or to accommodate the colliding entry in a separate table using an *overflow chaining* technique. We discuss these in the following paragraphs.

Rehashing

This technique uses a sequence of hashing functions $h_1, h_2, \ldots$ to resolve collisions. Let a collision occur while probing the table entry whose number is provided by $h_i(s)$. We use $h_{i+1}(s)$ to obtain a new entry number. This provides the new prediction in Algorithm 2.1. A popular technique called *sequential rehashing* uses the recurrence relation

$$h_{i+1}(s) \quad = \quad h_i(s) \bmod n + 1$$

to provide a series of hashing functions for rehashing.

A drawback of rehashing techniques is that a colliding entry accommodated elsewhere in the table may contribute to more collisions. This may lead to clustering of entries in the table.

Example 2.6 Let $h(\mathtt{a}) = h(\mathtt{b}) = h(\mathtt{c}) = 5$ and $h(\mathtt{d}) = 6$ in a language processor using sequential rehashing to handle collisions. If symbols are entered in a table in the sequence a, b, c, d, they would occupy the entries shown below:

Symbol	*Entry number*
a	5
b	6
c	7
d	8

While entering d, collisions occur in entries numbered 6 and 7 before d is accommodated in the 8th entry. Entries 5-8 form a cluster of size 4. A symbol x such that $h(\mathtt{x}) = 5$ suffers 4 collisions due to the cluster. Thus, the average number of collisions for a colliding entry = $\frac{1}{2} \times$(average size of cluster+1).

Table 2.1 summarizes the performance of sequential rehashing. For values of ρ in the range 0.6–0.8, performance of the hash table is quite adequate. p_s has values in the range of 1.75 to 3.0, while p_u has values in the range 2.5 to 5.0. This performance is obtained at the cost of over-commitment of memory for the symbol table. Taking $\rho = 0.7$ as a practical figure, the table must have $k_p^m/0.7$ entries, where k_p^m is the largest value of k_p in a practical mix of source programs. If a program contains less symbols,

the search performance would be better. However, a larger part of the table would remain unused. Note that unlike the sequential and binary search organizations, the hash table performance is independent of the size of the table; it is determined only by the value of ρ.

Table 2.1 Performance of sequential rehash

ρ	p_u	p_s
0.2	1.25	1.125
0.4	1.67	1.33
0.6	2.5	1.75
0.8	5.0	3.0
0.9	10.0	5.5
0.95	20.0	10.5

Hash table performance can be improved by reducing the clustering effect. Various rehashing schemes like sequential step rehash, quadratic and quadratic quotient rehash have been devised with this in view. Bell (1970) and Ackerman (1974) deal with these schemes. Price (1971) and Dhamdhere (1983) contain comprehensive treatments of hash table organizations.

Overflow chaining

Overflow chaining avoids the problems associated with the clustering effect (see Ex. 2.6) by accommodating colliding entries in a separate table called the *overflow table*. Thus, a search which encounters a collision in the primary hash table has to be continued in the overflow table. To facilitate this, a pointer field is added to each entry in the primary and overflow tables. The entry format is as follows:

Symbol	*Other info*	*Pointer*

A single hashing function h is used. All symbols which encounter a collision are accommodated in the overflow table. Symbols hashing into a specific entry of the primary table are chained together using the pointer field. On encountering a collision in the primary table, that is, in step 3 of Algorithm 2.3, one chain in the overflow table has to be searched.

Example 2.7 Figure 2.4 shows the organization of a hash table with overflow chaining. Symbols a, b, c and d of Ex. 2.6 are entered in the table as follows: a is entered in the 5th entry of the hash table. Symbol b collides with it. Hence b is put in the next available entry of the overflow table and the *pointer* field of the 5th entry of the hash table is set to point at it. c collides with a in the hash table and with b in the overflow table. A new entry is created for c and the *pointer* field of b's entry is set to point to it. d does not suffer a collision

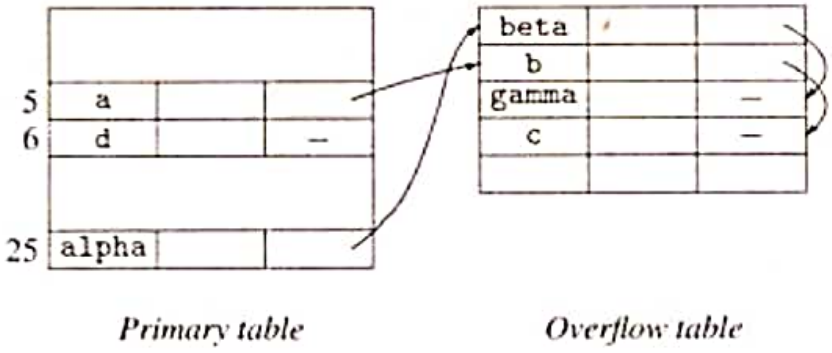


Fig. 2.4 Hash table organization with overflow chaining

Symbols `alpha`, `beta` and `gamma` form another chain since they all hash into entry 25 of the hash table.

The main drawback of the overflow chaining method is the extra memory requirement due to the presence of the overflow table, which must have $(k_p^m - 1)$ entries. An organization called scatter table organization is often used to reduce the memory requirements. In this organization, the hash table merely contains pointers, and all symbol entries are stored in the overflow table. Effectively, the hash table is merely a routing table (hence the name *scatter table*). Now the hash table should be large to ensure low values of ρ in it, and the overflow table needs to contain k_p^m entries.

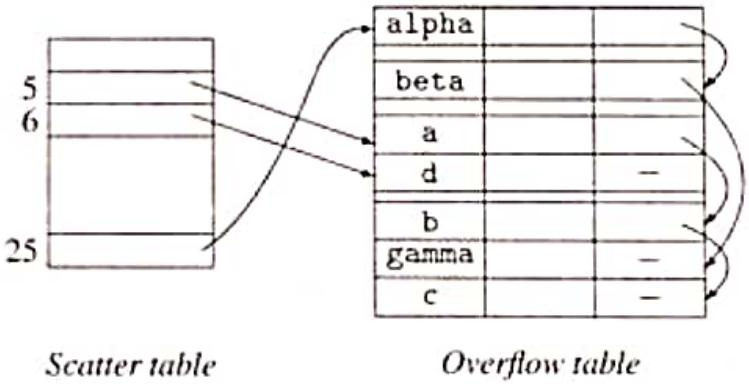


Fig. 2.5 Scatter table organization

Example 2.8 Figure 2.5 shows the scatter table organization for the symbols of Ex. 2.7. If each pointer requires 1 word and each symbol entry requires l words, the memory requirements are

$$
\begin{aligned}
&= \frac{k_p^m}{0.7} + k_p^m \times l \text{ words} \\
&= k_p^m \times (1.43 + l) \text{ words} \\
&\cong k_p^m \times l \text{ words}
\end{aligned}
$$

This compares favourably with the memory requirements if a rehashing technique was used to handle collisions, viz.

$$= (\frac{k_p^m}{0.7}) \times l \text{ words}$$
$$\cong 1.43\, k_p^m \times l \text{ words}$$

Memory requirements in the case of hash with overflow chaining (see Ex. 2.7) would have been

$$= (\frac{k_p^m}{0.7}) \times l + (k_p^m - 1) \times l \text{ words}$$
$$\cong 2.43\, k_p^m \times l \text{ words}$$

Note that both hash with overflow and scatter table organizations would perform better than the sequential rehash organization for the same value of ρ.

2.1.2 Linked List and Tree Structured Organizations

Linked list and tree structured organizations are nonlinear in nature, that is, elements of the search data structure are not located in adjoining memory areas. To facilitate search, each entry contains one or more pointers to other entries.

Symbol	*Other info*	*Pointer*	... *Pointer*

These organizations have the advantage that a fixed memory area need not be committed to the search structure. The system simply allocates a *header* element to start with. This element points to the first entry in the linked list or to the root of the tree. Other entries are allocated as and when needed.

Linked lists

Each entry in the linked list organization contains a single pointer field. The list has to be searched sequentially for obvious reasons. Hence its search performance is identical with that of sequential search tables, i.e. $p_s = l/2$ and $p_u = l$.

Binary trees

Each node in the tree is a symbol entry with two pointer fields—the *left_pointer* and the *right_pointer*. The following relation holds at every node of the tree: If s is the symbol in the entry, the left pointer points to a subtree containing all symbols $< s$ while the right pointer points to a subtree containing all symbols $> s$. This relation is used by the search procedure in a manner analogous to Algorithm 2.2. Algorithm 2.4 contains the search procedure. We use the notation $x.y$ to represent field y of node x and the notation $(p)^*.y$ to represent field y of the node pointed to by pointer p.

Algorithm 2.4 (Binary tree search)

1. *current_node_pointer* := address of root of the binary tree;

2. If s = (*current_node_pointer*)*.*symbol*, then exit with success;
3. If s < (*current_node_pointer*)*.*symbol* then
   current_node_pointer := (*current_node_pointer*)*.*left_pointer*;
   else *current_node_pointer* := (*current_node_pointer*)*.*right_pointer*;
4. If *current_node_pointer* = *nil* then
   exit with failure.
   else goto Step 2.

The best search performance is obtained when the tree is balanced. This performance is identical with that of the binary search table. In the worst case, the tree degenerates to a linked list and the search performance is identical with sequential search.

Example 2.9 Parts (a)-(d) of Figure 2.6 show various stages in the building of a binary tree when symbols are entered in the sequence p, c, t, f, h, k, e. Part (e) shows a balanced binary tree with the same symbols which gives better search performance.

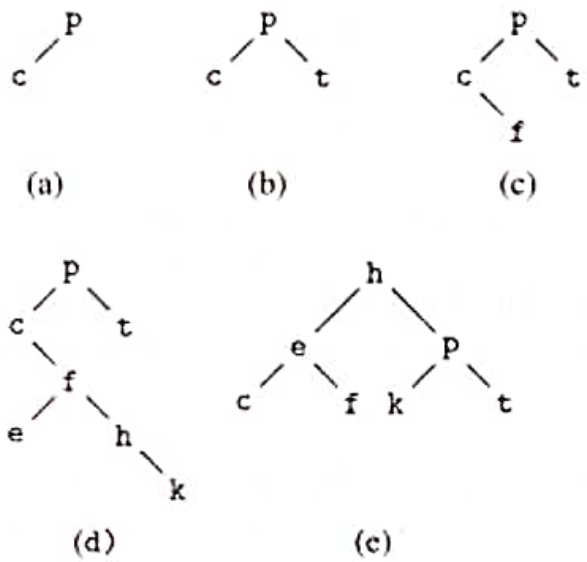


Fig. 2.6 Tree structured table organization

Nested search structures

Nested search structures are used when it is necessary to support a search along a secondary dimension within a search structure. Since the search structure cannot be linear in the secondary dimension, a linked list representation is used for the secondary search. Examples of nested search structures can be found in the handling of records of Pascal, PL/1, Cobol, etc., structures of C or handling of procedure parameters in any source language. Nested search structures are also known as *multi-list structures*.

Example 2.10 Figure 2.7 shows the symbol table entries for the Pascal record

```
personal_info   : record
                  name : array [1..10] of character;
                   sex : character;
                    id : integer;
                  end;
```

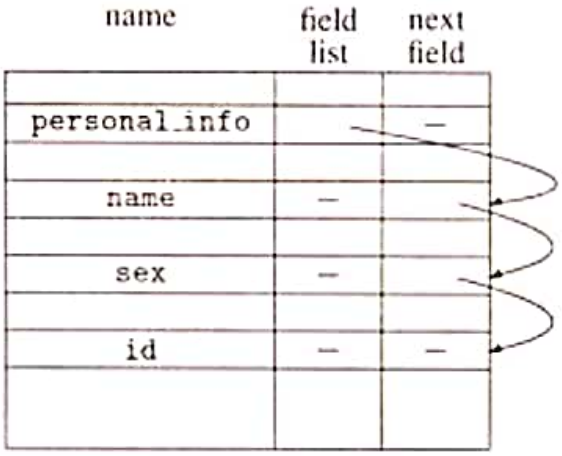


Fig. 2.7 Multi-list structure

Each symbol table entry contains two additional fields, *field list* and *next field*. *field list* of `personal_info` points to the entry for `name`. *next field* of `name` points to `sex`, etc. This organization permits the field `name` to be searched in the primary as well as secondary dimensions, i.e., as a symbol in the table as well as a field of `personal_info`. This enables efficient handling of occurrences like `name` and `personal_info.name` in the Pascal program.

EXERCISE 2.1

1. Comment on the feasibility of designing a hashing function to implement a direct entry organization for a fixed set of symbols. (Sprugnoli (1977) and Lyon (1978) report some related work).
2. An assembler is to be designed to handle programs containing up to 500 symbols. However, it is found that an average program contains only about 200 symbols. It is decided that a hash organization with sequential rehashing must be used. Evaluate the following three design alternatives in terms of memory requirements and access efficiency.

   (a) Size of the table = 550 entries
   (b) Size of the table = 700 entries
   (c) Size of the table = 500 entries

3. An assembler supports an option by which it produces an alphabetical listing of all symbols used in a program. Comment on the suitability of the following organizations

to organize the symbol table:

(a) Binary search organization
(b) Linked list organization
(c) Binary tree organization
(d) A multi-list organization

4. Many language processors use the concept of split-entry symbol tables. Each entry in such a table consists of two parts:

   (a) The fixed part containing the symbol field, a tag field, fields common to all variants, and a pointer field pointing to the variant part.
   (b) The variant part containing the fields relevant to the variant.

   Comment on the advantages of this organization.

5. Discuss the problem of deletion of entries in the following symbol table organizations:

   (a) Sequential search organization
   (b) Binary search organization
   (c) Hash table organizations with rehash and overflow chaining techniques
   (d) Linked list and tree structure organizations.

2.2 ALLOCATION DATA STRUCTURES

We discuss two allocation data structures, stacks and heaps.

2.2.1 Stacks

A *stack* is a linear data structure which satisfies the following properties:

1. Allocations and deallocations are performed in a *last-in-first-out* (LIFO) manner—that is, amongst all entries existing at any time, the first entry to be deallocated is the last entry to have been allocated.
2. Only the last entry is accessible at any time.

Figure 2.8 illustrates the stack model of allocation. Being a linear data structure, an area of memory is reserved for the stack. A pointer called the *stack base* (SB) points to the first word of the stack area. The stack grows in size as entries are created, i.e. as they are allocated memory. (We shall use the convention that a stack grows towards the higher end of memory. We depict this as downwards growth in the figures.) A pointer called the *top of stack* (TOS) points to the last entry allocated in the stack. This pointer is used to access the last entry. No provisions exist for access to other entries in the stack. When an entry is *pushed* on the stack (i.e. allocated in the stack), TOS is incremented by l, the size of the entry (we assume $l = 1$). When an entry is *popped*, i.e. deallocated, TOS is decremented by l (see Figs. 2.8(a)-(c)). To start with, the stack is *empty*. An empty stack is represented by TOS = SB − 1 (see Fig. 2.8(d)).

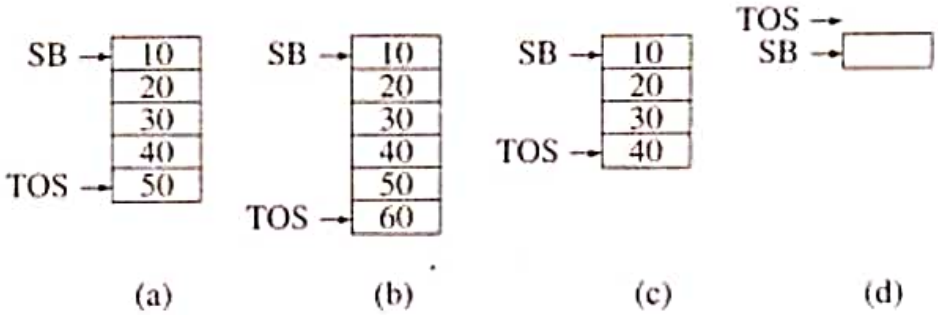


Fig. 2.8 Stack model of allocation

Extended stack model

The LIFO nature of stacks is useful when the lifetimes of the allocated entities follow the LIFO order. However, some extensions are needed in the simple stack model because all entities may not be of the same size. The size of an entity is assumed to be an integral multiple of the size of a stack entry. To allocate an entity, a *record* is created in the stack, where the record consists of a set of consecutive stack entries. For simplicity, the size of a stack entry, i.e. l, is assumed to be one word. Figure 2.9(a) shows the extended stack model. In addition to SB and TOS, two new pointers exist in the model:

1. A *record base pointer* (RB) pointing to the first word of the last record in stack.
2. The first word of each record is a *reserved pointer*. This pointer is used for housekeeping purposes as explained below.

The allocation and deallocation time actions in the extended stack model are described in the following paragraphs (see Fig. 2.9(b)–(c)).

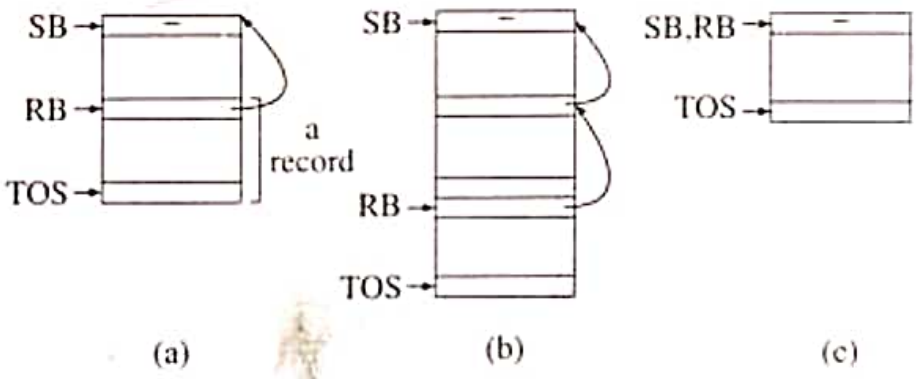


Fig. 2.9 Extended stack model

2.2.2 Heaps

A heap is a nonlinear data structure which permits allocation and deallocation of entities in a random order. An allocation request returns a pointer to the allocated area in the heap. A deallocation request must present a pointer to the area to be deallocated. The heap data structure does not provide any specific means to access an allocated entity. It is assumed that each user of an allocated entity maintains a pointer to the memory area allocated to the entity.

Example 2.12 Figure 2.11 shows the status of a heap after executing the following C program

```
float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc(5, sizeof(float));
floatptr2 = (float *) calloc(2, sizeof(float));
intptr = (int *) calloc(5, sizeof(int));
free(floatptr2);
```

Three memory areas are allocated by the calls on `calloc` and the pointers `floatptr1`, `floatptr2` and `intptr` are set to point to these areas. `free` frees the area allocated to `floatptr2`. This creates a 'hole' in the allocation. Note that following Section 2.1, each allocated area is assumed to contain a *length* field preceding the actual allocation.

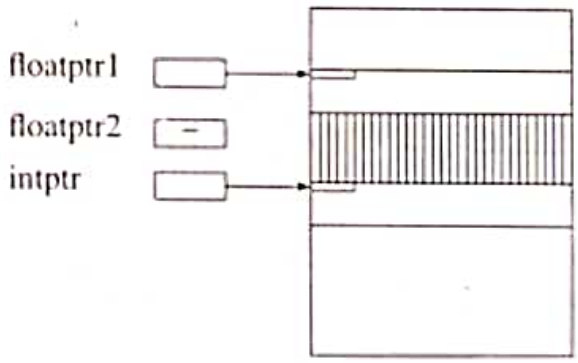


Fig. 2.11 Heap

Memory management

Example 2.12 illustrates how free areas (or 'holes') develop in memory as a result of allocations and deallocations in the heap. Memory management thus consists of identifying the free memory areas and reusing them while making fresh allocations. Speed of allocation/deallocation, and efficiency of memory utilization are the obvious performance criteria of memory management.

Identifying free memory areas

Two popular techniques used to identify free memory areas are:

1. Reference counts
2. Garbage collection.

In the reference count technique, the system associates a *reference count* with each memory area to indicate the number of its active users. The number is incremented when a new user gains access to that area and is decremented when a user finishes using it. The area is known to be free when its reference count drops to zero. The reference count technique is simple to implement and incurs incremental overheads, i.e. overheads at every allocation and deallocation. In the latter technique, the system performs garbage collection when it runs out of memory. Garbage collection makes two passes over the memory to identify unused areas. In the first pass it traverses all pointers pointing to allocated areas and *marks* the memory areas which are in use. The second pass finds all unmarked areas and declares them to be *free*. The garbage collection overheads are not incremental. They are incurred every time the system runs out of free memory to allocate to fresh requests.

To manage the reuse of free memory, the system can enter the free memory areas into a *free list* and service allocation requests out of the free list. Alternatively, it can perform *memory compaction* to combine these areas into a single free area.

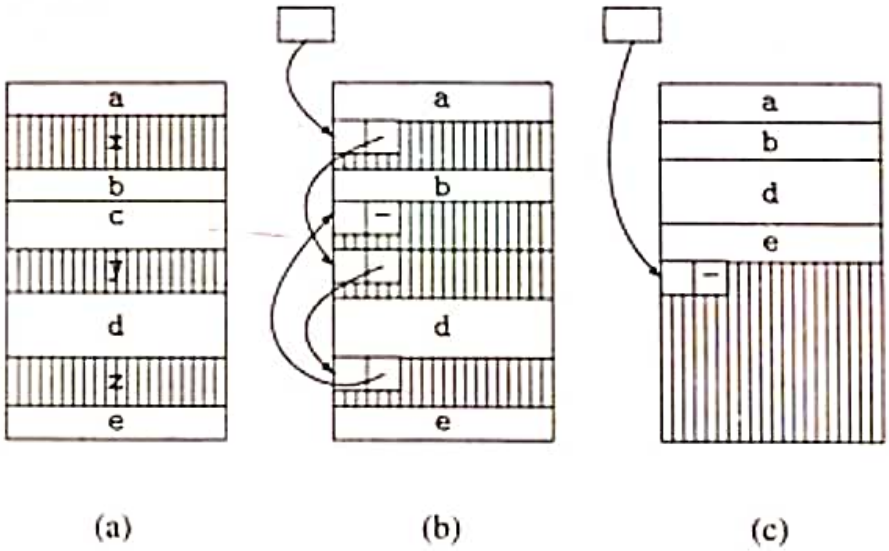


Fig. 2.12 (a) allocation status of heap, (b) free list, (c) after compaction

Example 2.13 Figure 2.12 shows free area management using free lists and memory compaction. Part (a) shows five areas named a-e in active use, and three free areas named x, y and z. The system has a *free area descriptor* permanently allocated. If a free list is to be used to facilitate memory allocation to fresh requests, the descriptor is used as a list header for the free list. The first word in each area is used to hold the count of words in the area and a pointer to the next free area in the list. Part (b) shows how area c is added to the free list consisting of x, y and z. If memory compaction is used, the *free area descriptor* describes the single free area resulting from com

paction. The first word in this area contains a count of words in the area and a *null* pointer. Figure 2.12(c) shows the results of memory compaction.

Reuse of memory

When memory compaction is used, fresh allocations are made from the block of free memory. The *free area descriptor* and the count of words in the free area are updated appropriately. When a free list is used, two techniques can be used to perform a fresh allocation:

1. First fit technique
2. Best fit technique.

The first fit technique selects the first free area whose size is $\geq n$ words, where n is the number of words to be allocated. The remaining part of the area is put back into the free list. This technique suffers from the problem that memory areas become successively smaller, hence requests for large memory areas may have to be rejected. The best fit technique finds the smallest free area whose size $\geq n$. This enables more allocation requests to be satisfied. However, in the long run it, too, may suffer from the problem of numerous small free areas.

Example 2.14 Let the free list consist of two areas called $area_1$ and $area_2$ of 500 words and 200 words, respectively. Let allocation requests for 100 words, 50 words and 400 words arise in the system. The first fit technique will allocate 100 words from $area_1$ and 50 words from the remainder of $area_1$. The free list now contains areas of 350 words and 200 words. The request for 400 words cannot be granted. The best fit technique will allocate 100 words and 50 words from $area_2$, and 400 words from $area_1$. This leaves areas of 100 words and 50 words in the free list.

Knuth (1973) discusses methods of overcoming the problems of first fit and best fit techniques. The *buddy* method may be used to merge adjoining free areas into larger free areas.

BIBLIOGRAPHY

(a) Data structures, general

1. Aho, A.V. and J.D. Ullman (1984): *Data Structures*, Addison-Wesley, New York.
2. Horowitz, E. and S. Sahni (1983): *Fundamentals of Data Structures*, Computer Science Press, California and Galgotia, New Delhi.
3. Knuth, D.E. (1985): *The Art of Computer Programming : Vol. 1*, Addison-Wesley, Reading and Narosa, New Delhi.
4. Wirth, N. (1976): *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs.

(b) Table management

Table management crucially determines the speed of language processing. Most books on compilers include a good coverage of table management techniques. Gries (1971) and

Dhamdhere (1983) contain comprehensive treatments of this topic. Price (1977) is an important review article on table management. Horowitz and Sahni (1983) and Tremblay and Sorenson (1983) are good sources on algorithms for table management.

1. Ackermann, A.F. (1974): "Quadratic search for hash tables of size p^n," *Commn. of ACM*, **17** (3), 164-165.
2. Bell, J.R. (1970): "The quadratic quotient method," *Commn. of ACM*, **13** (2), 107-109.
3. Dhamdhere, D.M. (1983): *Compiler Construction – Principles and Practice*, Macmillan India, New Delhi.
4. Donovan, J.J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
5. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.
6. Horowitz, E. and S. Sahni (1983): *Fundamentals of Data Structures*, Computer Science Press, California.
7. Knuth, D. (1973): *The Art of Computer Programming, Vol. III – Sorting and Searching*, Addison-Wesley, Reading.
8. Lyon, G. (1978): "Packed scatter tables," *Commn. of ACM*, **21** (10), 857-865.
9. Price, C.E. (1971): "Table lookup techniques," *Computing Surveys*, **3** (2), 49-65.
10. Sprugnoli, R. (1977): "Perfect hashing functions – a single probe retrieval method for static sets," *Commn. of ACM*, **20** (11), 841-850.
11. Tremblay, J.P. and P.G. Sorenson (1983): *An Introduction to Data Structures with Applications*, McGraw-Hill.

CHAPTER 3

Scanning & Parsing

3.1 SCANNING

Scanning is the process of recognizing the lexical components in a source string. As stated in Section 1.4.1.1, the lexical features of a language can be specified using Type-3 or regular grammars. This facilitates automatic construction of efficient recognizers for the lexical features of the language. In fact the scanner generator LEX generates such recognizers from the string specifications input to it (see Section 1.5.1).

In the early days of compilers, it was not possible to generate scanners automatically because the theory of PL grammars was not sufficiently understood and practiced. Specifications of lexical strings were excessively complex and scanners had to be hand coded to implement them. Fortran is notorious for very complex specifications of this kind.

Example 3.1 Successful scanning of some Fortran constructs requires interaction with the parser, e.g. consider the Fortran statements

```
DO 10 I = 1,2 and
DO 10 I = 1.2
```

The former is a DO statement while the latter is an assignment to a variable named `DO10I` (note that blanks are ignored in Fortran). Thus, scanning can only be performed after presence of the ',' identifies the former as a DO statement and its absence identifies the latter as an assignment statement. Fortunately, modern PL's do not contain such constructs.

Before proceeding with the details of scanning, it is important to understand the reasons for separating scanning from parsing. From Section 1.4.1.1, it is clear that each Type-3 production specifying a lexical component is also a Type-2 production. Hence it is possible to write a single set of Type-2 productions which specifies both lexical and syntactic components of the source language. However, a recognizer for

Type-3 productions is simpler, easier to build and more efficient during execution than a recognizer for Type-2 productions. Hence it is better to handle the lexical and syntactic components of a source language separately.

Finite state automata

Definition 3.1 (Finite state automaton (FSA)) *A* finite state automaton *is a triple* (S, Σ, T) *where*

- S *is a finite set of states, one of which is the* initial state s_{init}, *and one or more of which are the* final states
- Σ *is the alphabet of source symbols*
- T *is a finite set of state transitions defining transitions out of each* $s_i \in S$ *on encountering the symbols of* Σ.

We label the transitions of FSA using the following convention: A transition out of $s_i \in S$ on encountering a symbol $symb \in \Sigma$ has the label *symb*. We say a symbol *symb* is racognized by an FSA when the FSA makes a transition labelled *symb*. The transitions in an FSA can be represented in the form of a state transition table (STT) which has one row for each state $s_i \in S$ and one column for each symbol $symb \in \Sigma$. An entry STT$(s_i, symb)$ in the table indicates the id of the new state entered by the FSA if there exists a transition labelled *symb* in state s_i. If the FSA does not contain a transition out of state s_i for *symb*, we leave STT$(s_i, symb)$ blank. A state transition can also be represented by a triple (old state, source symbol, new state). Thus, the entry STT$(s_i, symb) = s_j$ and the triple $(s_i, symb, s_j)$ are equivalent.

The operation of an FSA is determined by its current state s_c. The FSA actions are limited to the following: Given a source symbol x at its input, it checks to see if STT(s_c, x) is defined—that is, if STT$(s_c, x) = s_j$, for some s_j. If so, it makes a transition to s_j, else it indicates an error and stops.

Definition 3.2 (DFA) *A* deterministic finite state automaton *(DFA) is an FSA such that* $t_1 \in T$, $t_1 \equiv (s_i, symb, s_j)$ *implies* $\nexists\, t_2 \in T$, $t_2 \equiv (s_i, symb, s_k)$.

Transitions in a DFA are deterministic—that is, at most one transition exists in state s_i for a symbol *symb*. Figure 3.1 illustrates the operation of a DFA. At any point in time, the DFA would have recognized some *prefix* α of the source string, possibly

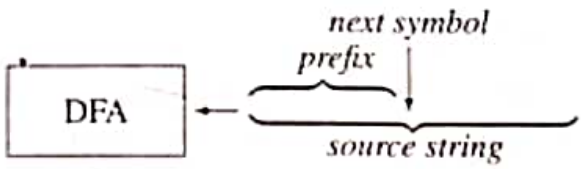


Fig. 3.1 Operation of a DFA

the null string, and would be poised to recognize the symbol pointed to by the pointer *next symbol*. The operation of a DFA is history-sensitive because its current state is a function of the prefix recognized by it. The DFA halts when all symbols in the source string are recognized, or an error condition is encountered. It can be seen that a DFA recognizes the longest valid prefix before stopping.

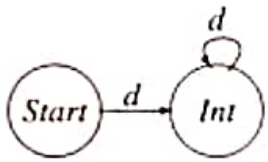
The validity of a string is determined by giving it at the input of a DFA in its initial state. The string is valid if and only if the DFA recognizes every symbol in the string and finds itself in a final state at the end of the string. This fact follows from the deterministic nature of transitions in the DFA.

Example 3.2 Figure 3.2 shows a DFA to recognize integer strings according to the Type-3 rule

<integer> ::= *d* | *<integer>* *d*

	Next Symbol
State	*d*
Start	*Int*
Int	*Int*

(a)



(b)

Fig. 3.2 DFA to recognise integer strings

where *d* represents a digit. Part (a) of the figure shows the STT for the DFA. Part (b) shows a diagrammatic representation of the states and state transitions. A transition from state s_i to state s_j on symbol *symb* is depicted by an arrow labelled *symb* from s_i to s_j. The initial and final states of the DFA are *Start* and *Int* respectively. Transitions during the recognition of string 539 are as given in the following table:

Current state	*Input symbol*	*New state*
Start	5	*Int*
Int	3	*Int*
Int	9	*Int*

The string leaves the DFA in state *Int* which is a final state, hence the string is a valid integer string. A string 5ab9 is invalid because no transition marked 'letter' exists in state *Int*.

Regular expressions

In Ex. 3.2 a single Type-3 rule was adequate to specify a lexical component. However, many Type-3 rules would be needed to specify complex lexical components like real constants. Hence we use a generalization of Type-3 productions called a *regular expression*.

Example 3.3 An organization uses an employee code which is obtained by concatenating the section id of an employee, which is alphabetic in nature, with a numeric code. The structure of the employee code can be specified as

<section code> ::= *l* | *<section code>* *l*
<numeric code> ::= *d* | *<numeric code>* *d*
<employee code> ::= *<section code>* *<numeric code>*

Note that a specification like

$$< s_code > \quad ::= \quad l \mid d \mid < s_code > l \mid < s_code > d$$

would be incorrect !

The regular expression generalizes on Type-3 rules by permitting multiple occurrences of a string form, and concatenation of strings. Table 3.1 shows regular expressions and their meanings.

Table 3.1 Regular expressions

Regular expression	*Meaning*
r	string r
s	string s
$r.s$ or rs	concatenation of r and s
(r)	same meaning as r
$r \mid s$ or $(r \mid s)$	alternation, i.e. string r or string s
$(r) \mid (s)$	alternation
$[r]$	an optional occurrence of string r
$(r)^*$	≥ 0 occurrences of string r
$(r)^+$	≥ 1 occurrences of string r

Example 3.4 The employee codes of Ex. 3.3 can be specified by the regular expression

$$(l)^+(d)^+$$

Some other examples of regular expressions are

integer	$[+ \mid -](d)^+$
real number	$[+ \mid -](d)^+.(d)^+$
real number with optional fraction	$[+ \mid -](d)^+.(d)^*$
identifier	$l(l \mid d)^*$

Table 3.2 Specification of a scanner

Regular expression	*Semantic actions*
$[+ \mid -](d)^+$	{Enter the string in the table of integer constants, say in entry n. Return the token $\boxed{Int\ \#n}$ }
$[+ \mid -]((d)^+.(d)^* \mid (d)^*.(d)^+)$	{Enter in the table of real constants. Return the token $\boxed{Real\ \#m}$ }
$l\,(l \mid d)^*$	{Compare with reserved words. If a match is found, return the token $\boxed{Kw\ \#k}$, else enter in symbol table and return the token $\boxed{Id\ \#i}$ }

EXERCISE 3.1

1. Develop regular expressions and DFAs for the following
   (a) a real number with optional integer and fraction parts,
   (b) a real number with exponential part,
   (c) a comment string in Pascal or C language.
2. Does the regular expression developed by you in problem 1 permit comments to be nested ? If not, make suitable changes.

3.2 PARSING

The goals of parsing are to check the validity of a source string, and to determine its syntactic structure. For an invalid string the parser issues diagnostic messages reporting the cause and nature of error(s) in the string. For a valid string it builds a parse tree to reflect the sequence of derivations or reductions performed during parsing. The parse tree is passed on to the subsequent phases of the compiler.

As described in Section 1.4.1, the fundamental step in parsing is to derive a string from an NT, or reduce a string to an NT. This gives rise to two fundamental approaches to parsing—*top down parsing* and *bottom up parsing*, respectively.

Parse trees and abstract syntax trees

A parse tree depicts the steps in parsing, hence it is useful for understanding the process of parsing. However, it is a poor intermediate representation for a source string because it contains too much information as far as subsequent processing in the compiler is concerned. An *abstract syntax tree* (AST) represents the structure of a source string in a more economical manner. The word 'abstract' implies that it is a representation designed by a compiler designer for his own purposes. Thus the

designer has total control over the information represented in an AST. It thus follows that an AST for a source string is not unique, whereas a parse tree is.

Example 3.6 Figure 3.4(a) shows a parse tree for the source string `a+b*c` according to Grammar (1.3). Figure 3.4(b) shows an AST for the same string. It contains sufficient information to represent the structure of the string. Note its economy compared to the parse tree.

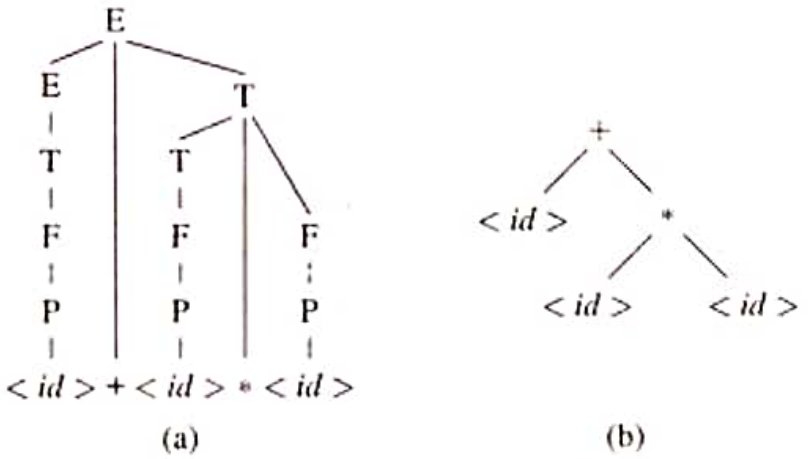


Fig. 3.4 Parse tree and AST

3.2.1 Top Down Parsing

Top down parsing according to a grammar G attempts to derive a string matching a source string through a sequence of derivations starting with the distinguished symbol of G. For a valid source string α, a top down parse thus determines a derivation sequence

$$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \alpha.$$

We shall identify the important issues in top down parsing by trying to develop a naive algorithm for it.

Algorithm 3.1 (Naive top down parsing)

1. *Current sentential form* (CSF) := 'S';
2. Let CSF be of the form $\beta A \pi$, such that β is a string of Ts (note that β may be null), and A is the leftmost NT in CSF. Exit with success if CSF = α.
3. Make a derivation $A \Rightarrow \beta_1 B \delta$ according to a production $A ::= \beta_1 B \delta$ of G such that β_1 is a string of Ts (again, β_1 may be null). This makes CSF = $\beta\beta_1 B \delta\pi$.
4. Go to Step 2.

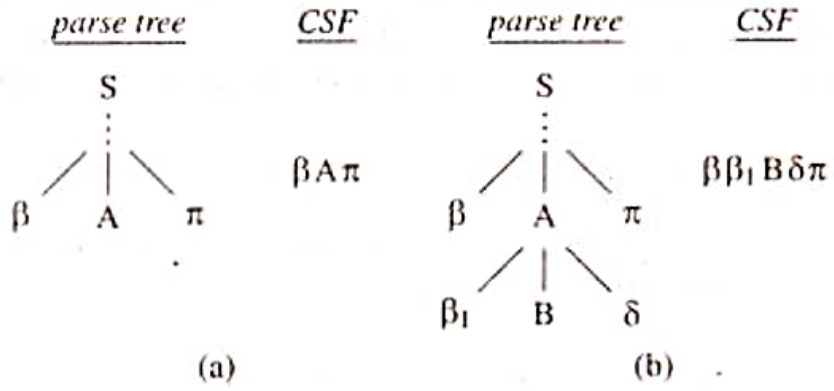


Fig. 3.5 Derivation $A \Rightarrow \beta_1 B\delta$ in top down parsing

Figure 3.5 depicts a step in top down parsing according to Algorithm 3.1. Since we make a derivation for the leftmost NT at any stage, top down parsing is also known as *left-to-left parsing* (LL Parsing).

Algorithm 3.1 lacks one vital provision from a practical viewpoint. Let CSF $\equiv \gamma C \delta$ with C as the leftmost NT in it and let the grammar production for C be

$$C ::= \rho \mid \sigma$$

where each of ρ, σ is a string of terminal and non-terminal symbols. Which RHS alternative should the parser choose for the next derivation ? The alternative we choose may lead us to a string of Ts which does not match with the source string α. In such a case, other alternatives would have to be tried out until we derive a sentence that matches the source string (i.e., a successful parse), or until we have systematically generated all possible sentences without obtaining a sentence that matches the source string (i.e., an unsuccessful parse). A naive approach to top down parsing would be to generate complete sentences of the source language and compare them with α to check if a match exists. However, this approach is inefficient for obvious reasons.

We introduce a check, called a *continuation check*, to determine whether the current sequence of derivations may be able to find a successful parse of α. This check is performed as follows: Let CSF be of the form $\beta A \pi$, where β is a string of n Ts. All sentential forms derived from CSF would have the form $\beta \ldots$. Hence, for a successful parse β must match the first n symbols of α. We can apply this check at every parsing step, and abandon the current sequence of derivations any time this condition is violated. The continuation check may be applied incrementally as follows: Let CSF $\equiv \beta A \pi$, then the source string must be $\beta \ldots$ (else we would have abandoned this sequence of derivations earlier). If the prediction for A is $A \Rightarrow \beta_1 B \delta$, where β_1 is a string of m terminal symbols, then the string β_1 must match m symbols following β in the source string (see Fig. 3.5). Hence we compare β_1 with m symbols to the right of β in the source string. This incremental check is more economical than a continuation check which compares the string $\beta\beta_1$ with $(n + m)$ symbols in the source string.

Predictions and backtracking

A typical stage in top down parsing can be depicted as follows:

$$\begin{array}{rcl} \text{CSF} & \equiv & \beta A \pi \\ & & \text{SSM} \\ & & \downarrow \\ \text{Source string} & : & \beta \ \mathtt{t} \end{array}$$

where CSF $\equiv \beta A \pi$ implies $S \overset{*}{\Rightarrow} \beta A \pi$ and the *source string marker* (SSM) points at the first symbol following β in the source string, i.e. at the terminal symbol 't'. We have already seen that if CSF = $\beta A \pi$, the source string must have the form $\beta \ldots$.

Parsing proceeds as follows: We identify the leftmost nonterminal in CSF, i.e. A. We now select an alternative on the RHS of the production for A. Since we do not know whether the derived string will satisfy the continuation check, we call this choice a *prediction*. The continuation check is applied incrementally to the terminal symbol(s), if any, occurring in the leftmost position(s) of the prediction. SSM is incremented if the check succeeds and parsing continues. If the check fails, one or more predictions are discarded and SSM is reset to its value before the rejected prediction(s) was made. This is called *backtracking*. Parsing is now resumed.

Implementing top down parsing

The following features are needed to implement top down parsing:

1. *Source string marker* (SSM): SSM points to the first unmatched symbol in the source string.
2. *Prediction making mechanism:* This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string in L_G can be derived from S.
3. *Matching and backtracking mechanism:* This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to by SSM. (This implements the incremental continuation check.) Backtracking is performed if the match fails. This involves resetting CSF and SSM to earlier values.

Continuation check and backtracking is performed in Step 3 of Algorithm 3.1. A complete algorithm incorporating these features can be found in (Dhamdhere, 1983).

Example 3.7 Lexically analysed version of the source string `a+b*c`, viz. $< id > + < id > * < id >$ is to be parsed according to the grammar

$$\begin{array}{lcl} E & ::= & T + E \mid T \\ T & ::= & V * T \mid V \\ V & ::= & < id > \end{array} \tag{3.1}$$

The prediction making mechanism selects the RHS alternatives of a production in a left-to-right manner. First few steps in the parse are:

1. SSM := 1; CSF := E;
2. Make the prediction E ⇒ T + E. Now, CSF = T + E.
3. Make the prediction T ⇒ V * T. CSF = V * T + E.
4. Make the prediction V ⇒ < *id* >. CSF = < *id* > * T + E. < *id* > matches with the first symbol of the source string. Hence, SSM := SSM + 1;
5. Match the second symbol of the prediction in Step 3, viz. '*'. This match fails, hence reject the prediction T ⇒ V * T. SSM and CSF are reset to 1 and T + E, respectively. (The situation now resembles that at the end of Step 2.)
6. Make a new prediction for T, viz. T ⇒ V. CSF = V + E.
7. Make the prediction V ⇒ < *id* >. CSF = < *id* > + E. < *id* > matches with the first symbol of the source string. Hence, SSM := SSM + 1;
8. Match the second symbol of the prediction in Step 2, viz. '+'. Match succeeds. SSM := SSM + 1;
9. Make the prediction E ⇒ T + E. CSF = < *id* > + T + E.
10. Make the prediction T ⇒ V * T. CSF = < *id* > + V * T + E.
11. ...

The predictions surviving at the end of the parse are listed in Table 3.3.

Table 3.3 Predictions in top sown parsing

Prediction	*Predicted Sentential Form*
E ⇒ T + E	T + E
T ⇒ V	V + E
V ⇒ < *id* >	< *id* > + E
E ⇒ T	< *id* > + T
T ⇒ V * T	< *id* > + V * T
V ⇒ < *id* >	< *id* > + < *id* > * T
T ⇒ V	< *id* > + < *id* > * V
V ⇒ < *id* >	< *id* > + < *id* > * < *id* >

Comments on top down parsing

Two problems arise due to the possibility of backtracking. First, semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a part of a successful parse. Second, precise error reporting is not possible. A mismatch merely triggers backtracking. A source string is known to be erroneous only after all predictions have failed. This makes it impossible to pinpoint the violations of PL specification.

Grammars containing left recursion are not amenable to top down parsing. For example, consider parsing of the string <*id*> + < *id* > * < *id* > according to the grammar

$$
\begin{aligned}
E &::= E + T \mid T \\
T &::= T * V \mid V \\
V &::= < id >
\end{aligned}
$$

The first prediction would be

$$E \Rightarrow E + T$$

which makes E the leftmost NT in CSF once again. Thus, the parser would enter an infinite loop of prediction making. To make top down parsing feasible, it is necessary to eliminate left recursion. This can be achieved by rewriting the grammar using right recursion as follows

$$
\begin{aligned}
E &::= T + E \mid T \\
T &::= V * T \mid V \\
V &::= < id >
\end{aligned}
\tag{3.2}
$$

However, this method is time-consuming and error-prone for large grammars. An alternative is to systematically eliminate left-recursion using the following rule: Rewrite a left recursive production

$$E ::= E + T \mid T$$

as

$$
\begin{aligned}
E &::= T\,E' \\
E' &::= +\,T\,E' \mid \varepsilon
\end{aligned}
\tag{3.3}
$$

The rationale for the rewriting is as follows: From the original production, it is clear that E produces a string consisting of one or more Ts separated by '+' symbols. Hence we write the production E ::= T E' with the expectation that E' will produce zero or more occurrences of the string '+ T'. This is what the production of E' achieves. After all left-recursive rules are converted in this manner, the resulting grammar can be used for top down parsing.

Top down parsing without backtracking

Elimination of backtracking in top down parsing would have several advantages—parsing would become more efficient, and it would be possible to perform semantic actions and precise error reporting during parsing. Prediction making becomes very crucial when backtracking is eliminated. The parser must use some contextual information from the source string to decide which prediction to make for the leftmost NT.

This is achieved as follows: If the leftmost NT is A and the source symbol pointed to by SSM is '`t`', parser selects that RHS alternative of A which can produce '`t`' as its first terminal symbol. An error is signalled if no RHS alternative can produce '`t`' as the first terminal symbol.

For the above approach to work, it is essential that at most one RHS alternative should be able to produce the terminal symbol '`t`' in the first position. The grammar may have to be modified to ensure this. As an example, consider parsing of the string $< id > + < id > * < id >$ according to Grammar (3.2). The first prediction is to be made using the production

$$E ::= T + E \mid T$$

such that the first terminal symbol produced by it would be $< id >$, the first symbol in the source string. From the grammar, we find that $T \Rightarrow V\ldots$ and $V \Rightarrow < id >$. Thus, any RHS alternative starting with a T can produce $< id >$ in the first position. However, both alternatives of E start with a T, so which one should the parser choose? To overcome this dilemma, we use *left-factoring* to ensure that the RHS alternatives will produce unique terminal symbols in the first position. The production for E is now rewritten as

$$\begin{aligned} E &::= T\,E'' \\ E'' &::= +\,E \mid \varepsilon \end{aligned}$$

The first prediction according to this grammar is $E \Rightarrow T\ E''$ since the first source symbol is '$<id>$'. When E'' becomes the leftmost NT in CSF, we would make the prediction $E'' \Rightarrow +\ E$ if the next source symbol is '+' and the prediction $E'' \Rightarrow E$ in all other cases. If the next source symbol is '+', we would make the prediction $E'' \Rightarrow +\ E$, else we would make the prediction $E'' \Rightarrow \varepsilon$. Thus, parsing is no longer by trial-and-error. Such parsers are known as *predictive parsers*.

The complete rewritten form of Grammar (3.2) is

$$\begin{aligned} E &::= T\,E'' \\ E'' &::= +\,E \mid \varepsilon \\ T &::= V\,T'' \\ T'' &::= *\,T \mid \varepsilon \\ V &::= < id > \end{aligned} \tag{3.4}$$

Note that grammar (3.3) does not need left factoring since its RHS alternatives produce unique terminal symbols in the first position.

Example 3.8 Parsing of $< id > + < id > * < id >$ according to Grammar (3.4) proceeds as shown in Table 3.4. Note that *Symbol* is the symbol pointed to by SSM.

To start with, CSF = E and SSM points to '$<id>$'. The first three steps are obvious. In the 4th step, SSM points to '+' and the leftmost NT is T''. This leads to the prediction $T'' \Rightarrow \varepsilon$.

Table 3.4 Top down parsing without backtracking

Sr. No.	*CSF*	*Symbol*	*Prediction*
1.	E	$< id >$	$E \Rightarrow T\,E''$
2.	$T\,E''$	$< id >$	$T \Rightarrow V\,T''$
3.	$V\,T''E''$	$< id >$	$V \Rightarrow < id >$
4.	$< id > T''E''$	+	$T'' \Rightarrow \varepsilon$
5.	$< id > E''$	+	$E'' \Rightarrow + E$
6.	$< id > + E$	$< id >$	$E \Rightarrow T\,E''$
7.	$< id > + T\,E''$	$< id >$	$T \Rightarrow V\,T''$
8.	$< id > + V\,T''E''$	*id*	$V \Rightarrow < id >$
9.	$< id > + < id > T''E''$	*	$T'' \Rightarrow * T$
10.	$< id > + < id > * T\,E''$	$< id >$	$T \Rightarrow V\,T''$
11.	$< id > + < id > * V\,T''E''$	$< id >$	$V \Rightarrow < id >$
12.	$< id > + < id > * < id > T''E''$	–	$T'' \Rightarrow \varepsilon$
13.	$< id > + < id > * < id > E''$	–	$E'' \Rightarrow \varepsilon$
14.	$< id > + < id > * < id >$	–	–

3.2.1.1 Practical Top Down Parsing

A recursive descent parser

A recursive descend (RD) parser is a variant of top down parsing without backtracking. It uses a set of recursive procedures to perform parsing. Salient advantages of recursive descent parsing are its simplicity and generality. It can be implemented in any language supporting recursive procedures.

To implement recursive descent parsing, a left-factored grammar is modified to make repeated occurrences of strings more explicit. Grammar (3.4) is rewritten as

$$\begin{aligned} E &::= T\,\{+\,T\}^* \\ T &::= V\,\{*\,V\}^* \\ V &::= < id > \end{aligned} \tag{3.5}$$

where the notation $\{..\}^*$ indicates zero or more occurrences of the enclosed specification. A parser procedure is now written for each NT of G. It handles prediction making, matching and error reporting for that NT. The structure of a parser procedure is dictated by the grammar production for the NT. If A ::= ..B.. is a production of G, the parser procedure for A contains a call on the procedure for B.

Example 3.9 A skeletal recursive descent parser for Grammar (3.5) is shown in Fig. 3.6. The parser returns an AST for a valid source string, and reports an error for an invalid string. The procedures `proc_E`, `proc_T` and `proc_V` handle the parsing for E, T and

```
procedure proc_E : (tree_root);
   /* This procedure constructs an AST for 'E'
      and returns a pointer to its root */
   var
      a, b : pointer to a tree node;
   begin
      proc_T (a);
         /* Returns a pointer to the root of tree for T */
      while (nextsymb = '+') do
         match ('+');
         proc_T (b);
         a := treebuild ('+', a, b);
               /* Builds an AST and returns pointer
                  to its root */
      tree_root := a;
      return;
end proc_E;

procedure proc_T (tree_root);
   var
      a, b : pointer to a tree node;
   begin
      proc_V (a);
      while (nextsymb = '*') do;
         match ('*');
         proc_V (b);
         a := treebuild ('*', a, b);
      tree_root := a;
      return;
end proc_T;

procedure proc_V (tree_root);
   var
      a : pointer to a tree node;
   begin
      if (nextsymb = < id >) then
         tree_root := treebuild (< id >, -, -);
         else print "Error !";
      return;
end proc_V;
```

Fig. 3.6 A recursive descent parser

V, respectively, and build ASTs for these NTs using the procedure `treebuild`. Procedure `match` increments SSM. Procedure `proc_E`, the procedure for E, always calls `proc_T` to perform parsing and AST building for a T. If the next source symbol (which is assumed to be contained in the parser variable `nextsymb`) is a '+', then another T is expected. Hence `proc_T` is called again. This process is repeated until the symbol following a T is not a '+'. At this point, `proc_E` returns to the parser control routine. The control routine should now check whether the entire source string has been successfully parsed, else indicate an error. Error detection and reporting is restricted to those parser procedures which produce terminal symbol(s) in the first position. For grammar (3.5) only the routine for V would declare error if anything but an $< id >$ is encountered. The reader can verify that error indication would be precise.

Note that the parser procedures do not call themselves recursively. However, indirect recursion would result if the grammar specification contains indirect recursion—for example, if a rule like

$$V ::= < id > \mid (E)$$

existed in grammar (3.5).

An LL(1) parser

An LL(1) parser is a table driven parser for left-to-left parsing. The '1' in LL(1) indicates that the grammar uses a look-ahead of one source symbol—that is, the prediction to be made is determined by the next source symbol. A major advantage of LL(1) parsing is its amenability to automatic construction by a parser generator.

Figure 3.7 shows the parser table for an LL(1) parser for Grammar (3.6).

$$\begin{array}{lcl} E & ::= & T\,E' \\ E' & ::= & +\,T\,E' \mid \varepsilon \\ T & ::= & V\,T' \\ T' & ::= & *\,V\,T' \mid \varepsilon \\ V & ::= & < id > \end{array} \qquad (3.6)$$

The parsing table (PT) has a row for each NT $\in$ SNT and a column for each T $\in \Sigma$. A parsing table entry PT (nt_i, t_j) indicates what prediction should be made if nt_i is the leftmost NT in a sentential form and t_j is the next source symbol. A blank entry in PT indicates an error situation. A source string is assumed to be enclosed between the symbols '$\vdash$' and '$\dashv$'. Hence the parser starts with the sentential form $\vdash$ E $\dashv$.

Example 3.10 The sequence of predictions made by the parser of Fig. 3.7 for the source string

$$\vdash < id > + < id > * < id > \dashv$$

can be seen in Table 3.5 where *symbol* is the next source symbol. At every stage the parser refers to the table entry PT (nt_i, t_j) where nt_i is the leftmost nonterminal in the string and t_j is the next source symbol. Note that these steps are equivalent to the steps in Ex. 3.8.

Non-terminal	*Source symbol*			
	<*id*>	+	*	⊣
E	E ⇒ TE′			
E′		E′ ⇒ +TE′		E′ ⇒ ε
T	T ⇒ VT′			
T′		T′ ⇒ ε	T′ ⇒ *VT′	T′ ⇒ ε
V	V ⇒ <*id*>			

Fig. 3.7 LL(1) parser table

Table 3.5 LL parsing

Current sentential form	*Symbol*	*Prediction*
⊢ E ⊣	< *id* >	E ⇒ TE′
⊢ TE′ ⊣	< *id* >	T ⇒ VT′
⊢ VT′E′ ⊣	< *id* >	V ⇒ < *id* >
⊢ < *id* > T′E′ ⊣	+	T′ ⇒ ε
⊢ < *id* > E′ ⊣	+	E′ ⇒ + TE′
⊢ < *id* > + TE′ ⊣	< *id* >	T ⇒ VT′
⊢ < *id* > + VT′E′ ⊣	< *id* >	V ⇒ < *id* >
⊢ < *id* > + < *id* > T′E′ ⊣	*	T′ ⇒ *VT′
⊢ < *id* > + < *id* > * VT′E′ ⊣	< *id* >	V ⇒ < *id* >
⊢ < *id* > + < *id* > * < *id* >T′E′ ⊣	⊣	T′ ⇒ ε
⊢ < *id* > + < *id* > * < *id* >E′ ⊣	⊣	E′ ⇒ ε
⊢ < *id* > + < *id* > * < *id* > ⊣	–	–

The procedure for the construction of the parser table is described in the literature cited at the end of the chapter.

EXERCISE 3.2.1

1. Study the operation of the recursive descent parser described in this section for the following input strings:

   (a) < *id* > + < *id* > * < *id* >

   (b) < *id* > * < *id* >< *id* >

   Verify that precise error indication is provided by the parser.

2. A recursive descent parser is to be developed for Grammar (1.3).

(a) What changes will you make in the grammar to make it suitable for recursive descent parsing?

(b) Code the parser procedures.

(c) Identify the procedures that perform error indication.

3. Compare and contrast recursive descent parsing with LL(1) parsing. What grammar forms are acceptable to each of these?

3.2.2 Bottom Up Parsing

A bottom up parser constructs a parse tree for a source string through a sequence of reductions. The source string is valid if it can be reduced to S, the distinguished symbol of G. If not, an error is to be detected and reported during the process of reduction. Bottom up parsing proceeds in a left-to-right manner, i.e. attempts at reduction start with the first symbol(s) in the string and proceed to the right. A typical stage during bottom up parsing is depicted as follows:

$$\underbrace{\ldots}_{\text{string of NTs \& Ts}} \overset{\text{SSM}}{\downarrow} \underbrace{\ldots}_{\text{string of Ts}}$$

Reductions have been applied to the string on the left of SSM. Hence this string is composed of NTs and Ts. Remainder of the string, yet to be processed by the parser, consists of Ts alone.

We try the following naive approach to bottom up parsing: Let there be n symbols to the left of SSM in the current string form. We try to reduce some part of the string to the immediate left of SSM. Let this part have r symbols in it, and let it be reduced to the NT A. This reduction can be depicted as follows:

$$\text{Before reduction :}\quad \overbrace{\underbrace{\ldots}_{r \text{ symbols}}}^{n \text{ symbols}} \overset{\text{SSM}}{\downarrow} \ldots$$

$$\text{After reduction :}\quad \overbrace{\qquad}^{n-r \text{ symbols}} \text{A} \overset{\text{SSM}}{\downarrow} \ldots$$

Since we do not know the value of r, we try the values $r = n$, $r = n - 1$, ... $r = 1$. This approach is summarized in Algorithm 3.2.

Algorithm 3.2 (Naive bottom up parsing)

1. SSM := 1; n := 0;
2. $r := n$;
3. Compare the string of r symbols to the left of SSM with *all* RHS alternatives in G which have a length of r symbols.

4. If a match is found with a production A ::= α, then
   reduce the string of r symbols to the NT A.
   $n := n - r + 1$;
   Go to Step 2;
5. $r := r - 1$;
   If $r > 0$, go to Step 3;
6. If no more symbols exist to the right of SSM then
   if current string form = 'S' then
   exit with success
   else report error and exit with failure
7. SSM := SSM + 1;
   $n := n + 1$;
   Go to Step 2;

Thus the parser makes as many reductions as possible at the current position of SSM (see Step 5). When no reductions are possible at the current position, SSM is incremented by one symbol (see Step 7). This is called a *shift* action. The parsing process thus consists of shift and reduce actions applied in a left-to-right manner. Hence bottom up parsing is also known as *LR parsing* or *shift-reduce parsing*.

Algorithm 3.2 is unsatisfactory for two reasons. First, it is inefficient due to the large number of comparisons made in Step 3. Second, it performs reductions in a manner which may conflict with operator priorities. For example, consider Grammar (1.3) and the input string $< id > + < id > * < id >$. The correct parse tree is as shown in Fig. 3.8(b), where the symbols E, T, F and P stand for *<exp>*, *<term>*, *<factor>* and *<primary>*, respectively. The parser of Algorithm 3.2 builds the parse tree of Fig. 3.8(a) by erroneously reducing $< id > + < id >$ to an E. Further, since E * T does not appear on the RHS of any production, the parser indicates an error. However, the string is valid according to Grammar (1.3). This difficulty arises due to the premature reduction of $< id > + < id > \rightarrow$ E performed by Algorithm 3.2.

To overcome this problem, we need a criterion which will indicate when to perform a reduction and when not to, viz. given the current stage of parsing

$$\begin{array}{c} \text{SSM} \\ \downarrow \\ \ldots \alpha t \gamma \ldots \end{array}$$

the criterion should indicate which of the following should be performed:

1. Apply a reduction involving α, and possibly some symbols to its left, to obtain an NT A.
2. Perform a shift action and now apply a reduction to some string $\delta\alpha t$ or σt to obtain an NT B. (Here σ is a substring of α.)
3. Perform one or more shift actions and apply a reduction to the string $t \ldots$ to obtain an NT C. Now apply a reduction to some string $\delta\alpha C$ or σC.

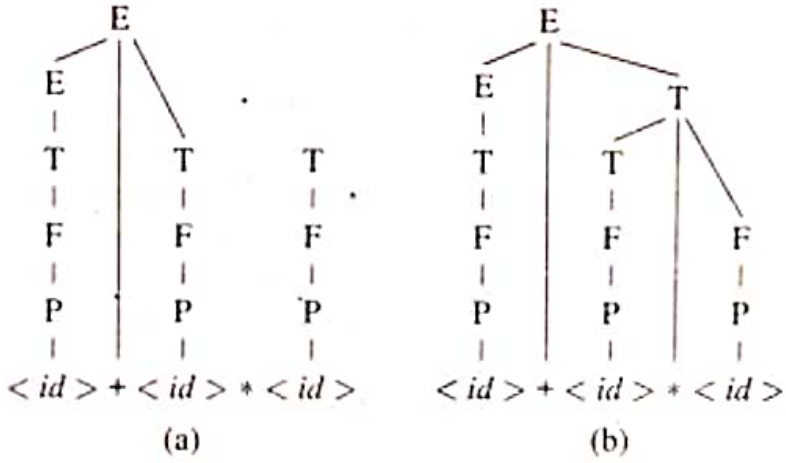


Fig. 3.8 Parse trees in bottom up parsing

In other words, given the sentential form ... $\alpha t \gamma$ ..., the criterion should indicate whether to reduce (a substring of) α before, along with, or after some string t... appearing to its right. Such a criterion is provided by the notion of *precedence* of grammar symbols.

Simple precedence

Definition 3.3 (Simple precedence) *A grammar symbol* a precedes *symbol* b, *where each of* a, b *is a T or NT of G, if in a sentential form* ... a b ..., a *should be reduced prior to* b *in a bottom up parse.*

We use the notation a ·> b for the words 'a precedes b'. Two possibilities arise if a does not precede b,

1. b precedes a, (i.e. b should be reduced prior to a), represented as b ·> a, or
2. a and b have equal precedence (i.e. a and b should be reduced in the same step), represented as a = b.

Note that a precedence relation is defined between grammar symbols a and b only if a and b can occur side by side in a sentential form. We use this fact to obtain the precedence relation between a and b as follows:

1. Consider some sentential form ... a b ... .
2. Determine the sequence of derivations $S \overset{*}{\Rightarrow}$ ... a b ... such that the last derivation derives a string containing a or b or both. Number the derivations in this sequence from 1 to q.
3. Consider the derivation numbered q. This is the last derivation for obtaining ... a b ... . Let this be of the form $A \Rightarrow \beta$. Then $\beta \rightarrow A$ must be the first reduction in the bottom up parse of the string ... a b ... . Now (3.7)
   (a) a ·> b if $\beta \equiv$ ...a
   (b) a <· b if $\beta \equiv$ b...
   (c) a = b if $\beta \equiv$ ...a b... .

Definition 3.4 (Simple precedence grammar (SPG))

Grammar G is a simple precedence grammar *if for all terminal and nonterminal symbols* a, b *of G, a unique precedence relation exists for* a, b.

Using the notion of precedence in an SPG, we can identify a unit for reduction in a sentential form by looking for the precedence relations

$$<\cdot\, s_1 \doteq s_2 \doteq \ldots \doteq s_r \,\cdot>$$

in the source string where each s_i is a T or NT. Here s_1, s_r are the first and last symbols to participate in the reduction. Needless to say that the string $s_1 s_2 \ldots s_r$ would form some RHS alternative in the grammar.

We will now formally state the problem of bottom up parsing as follows:

Definition 3.5 (Simple phrase) α *is a* simple phrase *of the sentential form* $\ldots\alpha\beta\ldots$ *if there exists a production of the grammar* A ::= α *and* $\alpha \rightarrow$ A *is a reduction in the sequence of reductions* $\ldots\alpha\beta\ldots \rightarrow \ldots \rightarrow$ S.

Definition 3.6 (Handle) *A* handle *of a sentential form is the left-most simple phrase in it.*

Example 3.11 E + T is not a simple phrase of the sentential form E + T * F according to grammar (1.3) since its reduction to E does not lead to the distinguished symbol. However, T * F is a simple phrase of the sentential form (see Fig. 3.8).

The deficiency of Algorithm 3.2 lies in its failure to identify simple phrases and handles. Algorithm 3.3 rectifies this deficiency.

Algorithm 3.3 (Bottom up parsing)

1. Identify the handle of the current string form.
2. If a handle exists, reduce it. Go to Step 1.
3. If the current string form = 'S' then exit with success
   else report error and exit with failure.

In practice, most grammars are not SPGs. Hence we will illustrate bottom up parsing for the class of operator grammars.

Example 3.12 Grammar (1.3) is not an SPG because of the following: In the string E + T * F, + $<\cdot$ T due to the production T ::= T * F. However, in E + T, + $\doteq$ T due to the production E ::= E + T.

Operator precedence grammars

In Section 1.4.1.1, we defined an operator grammar as a grammar none of whose productions contain two NTs appearing side-by-side. By induction, NTs cannot appear side-by-side in any sentential form of an operator grammar. Hence it is possible to ignore the presence of NTs and define precedence relations between operators for making parsing decisions. An operator precedence grammar (OPG) is an operator grammar in which the precedences between operators are unique. Such grammars typically arise in expressions.

Operator precedence

Precedence between *operators* a and b appearing in a sentential form ...aPb... where P is an NT or a null string, is termed *operator precedence*. The rules of procedure (3.7) can be used to determine operator precedences by considering the sentential form ...aPb... instead of the form ...ab.... Alternatively, it is possible to determine precedence relations from the productions of G as follows

1. a $\doteq$ b iff there exists a grammar production
   C ::= βabγ or C ::= βaAbγ.
2. a $\cdot>$ b iff there exist grammar productions
   C ::= βAbγ and (3.8)
   A ::= πa | πaD
3. a $<\cdot$ b iff there exist grammar productions
   C ::= βaBγ and
   B ::= bδ | Dbδ

An *operator precedence matrix* (OPM) represents operator precedence relations between pairs of operators. The entry OPM (a, b) represents the precedence of operator a with respect to operator b in a sentential form ...aPb... , where P may be a null string.

Example 3.13 The precedence relations for the grammar

$$\begin{array}{lcl} E & ::= & E + T \mid T \\ T & ::= & T * V \mid V \\ V & ::= & < id > \end{array}$$

are shown in Fig. 3.9. The entry OPM (+, *) = $<\cdot$, represents the precedence of '+' as a left operator and '*' as the right operator of a pair of adjoining operators in a string, e.g. '+' and '*' of the string $< id > + < id > * < id >$. This precedence relation is obtained by applying procedure (3.7) to the sequence of derivations E $\Rightarrow$ E + T, T $\Rightarrow$ T * V, or by applying rule 3 of the rules (3.8) to the same productions.

There is a simpler way to obtain the precedence relations which works well for most grammars. This is based on the notions of associativity and relative priority of

LHS operator	RHS operator	
	+	*
+	·>	<·
*	·>	·>

Fig. 3.9

operators. A high priority operator always precedes a low priority operator appearing to its left or right. When two occurrences of an operator occupy adjoining positions of a string, the left occurrence precedes the right occurrence if the operator is left associative, else the right occurrence precedes the left occurrence.

Example 3.14 Consider the following grammar

$$\begin{array}{lcl} S & ::= & \vdash E \dashv \\ E & ::= & E + T \mid T \\ T & ::= & T * V \mid V \\ V & ::= & < id > \mid (E) \end{array} \qquad (3.9)$$

Figure 3.10 shows the operator precedence matrix for the grammar. The entry OPM ('+', '+') = ·> follows from the fact that '+' is left associative. The entries for '(' and ')' can be explained as follows: Consider an expression string

... b * (c + d) * e ...

Since a parenthesized expression is to be evaluated before its left and right neighbours, any operator would have <· relation with '(' appearing to its right, viz. '*' <· '(' in the above string, and '(' would have a <· relation with any operator appearing to its right, viz. '(' <· '+'. Similarly, any operator would have ·> relation with ')' appearing to its right, while ')' would have a ·> relation with any operator appearing to its right. This is seen from the fact that '+' ·> ')', while ')' ·> '*' in the above string. When '(' and ')' occur side by side, they would have ≐ precedence with one another. In fact, this is the only instance of equal precedence in expressions. Same precedence relations would be obtained by applying procedure (3.7) or rules (3.8).

Note that '⊣' cannot be the LHS operator of a pair. Similarly '⊢' cannot be the RHS operator of a pair. Hence no rows and columns exist for these operators in OPM, respectively. The entries OPM ('(', '⊣') and OPM ('⊢', ')') are blank. Both these entries represent erroneous combinations of operators.

Operator precedence parsing

Example 3.15 Consider parsing of the string

$$\vdash < id > + < id > * < id > \dashv$$

LHS operator	RHS operator				
	+	*	(	)	⊣
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	≐	
)	·>	·>		·>	·>
⊢	<·	<·	<·		≐

Fig. 3.10 Operator precedence matrix

according to Grammar (3.9). The precedence relations from Figure 3.10 are marked below the string

Initial string : ⊢ <*id*> + <*id*> * <*id*> ⊣

<· <· ·>

The first <· is the precedence relation ⊢ <· +, the second <· represents + <· * etc. The handle is the string consisting of '* and its operands. After its reduction, the reduced string is

Reduced string : ⊢ <*id*> + ... ⊣

<· ·>

which leads to reduction of '+'.

Since operator precedence parsing ignores the NTs in a string, it is not easy to build a parse tree for a source string. However, an AST can be built easily. To develop an algorithm for AST building, we begin with some terminology and observations. We call the operator pointed to by SSM as the *current operator* and the operator to its left as the *previous operator*. From Ex. 3.15 it is clear that the previous operator must be reduced if previous operator ·> current operator, else a shift action must be performed. Thus end of the handle is known when previous operator ·> current operator. To find the beginning of the handle, we use the fact that the only instance of ≐ is '(' ≐ ')'. Hence the handle merely consists of the previous operator and its operands unless the previous operator is '('. If the previous operator is '(', the handle consists of '(..)'.

From these observations it is clear that apart from the current operator, only the previous operator needs to be considered at any point. A stack is therefore an appropriate data structure to use. An operator is pushed on the stack during a shift action and popped off during a reduce action. We can accommodate the right operand of an operator in the stack entry of the operator. The left operand, if present, would exist in the previous stack entry. We refer to the stack entry below the TOS entry as (TOS-1) entry, or TOSM entry for short.

Algorithm 3.4 (Operator Precedence Parsing)

Data structures

Stack : Each stack entry is a record with two fields, *operator* and *operand_pointer*.

Node : A *node* is a record with three fields, *symbol*, *left_pointer*, and *right_pointer*.

Functions

newnode (*operator*, *l_operand_pointer*, *r_operand_pointer*) creates a *node* with appropriate pointer fields and returns a pointer to the node.

1. TOS := SB − 1; SSM := 0;
2. Push '⊢' on the stack.
3. SSM := SSM + 1;
   If current source symbol is an operator, then go to Step 5.
4. *x* := *newnode* (*source symbol*, *null*, *null*);
   TOS.*operand_pointer* := *x*;
   Go to Step 3;
5. While TOS operator ·> current operator
   x := *newnode* (TOS *operator*, TOSM.*operand_pointer*,
   TOS.*operand_pointer*);
   Pop an entry off the stack.
   TOS.*operand_pointer* := *x*;
6. If TOS operator <· current operator, then
   Push the current operator on the stack.
   Go to Step 3;
7. If TOS operator = current operator, then
   If TOS operator = '⊢', then exit successfully.
   If TOS operator = '(', then
   temp := TOS.*operand_pointer*;
   Pop an entry off the stack.
   TOS.*operand_pointer* := *temp*;
   Go to Step 3;
8. If no precedence defined between TOS operator and current operator then
   Report error and exit unsuccessfully.

Example 3.16 Consider parsing of the string

$$\vdash < id >_a + < id >_b * < id >_c \dashv$$

according to grammar (3.9), where $< id >_a$ represents a. Figure 3.11 shows steps in its parsing. Figures 3.11(a)-3.11(c) show the stack and the AST when current operator is '+', '*' and '⊣', respectively. In Fig. 3.11(c), TOS operator ·> current operator.

This leads to reduction of '*' (see Step 5 of Algorithm 3.4). Figure 3.11(d) shows the situation after the reduction. The new TOS operator, i.e. '+', -> current operator. This leads to reduction of '+' as shown in Fig. 3.11(e).

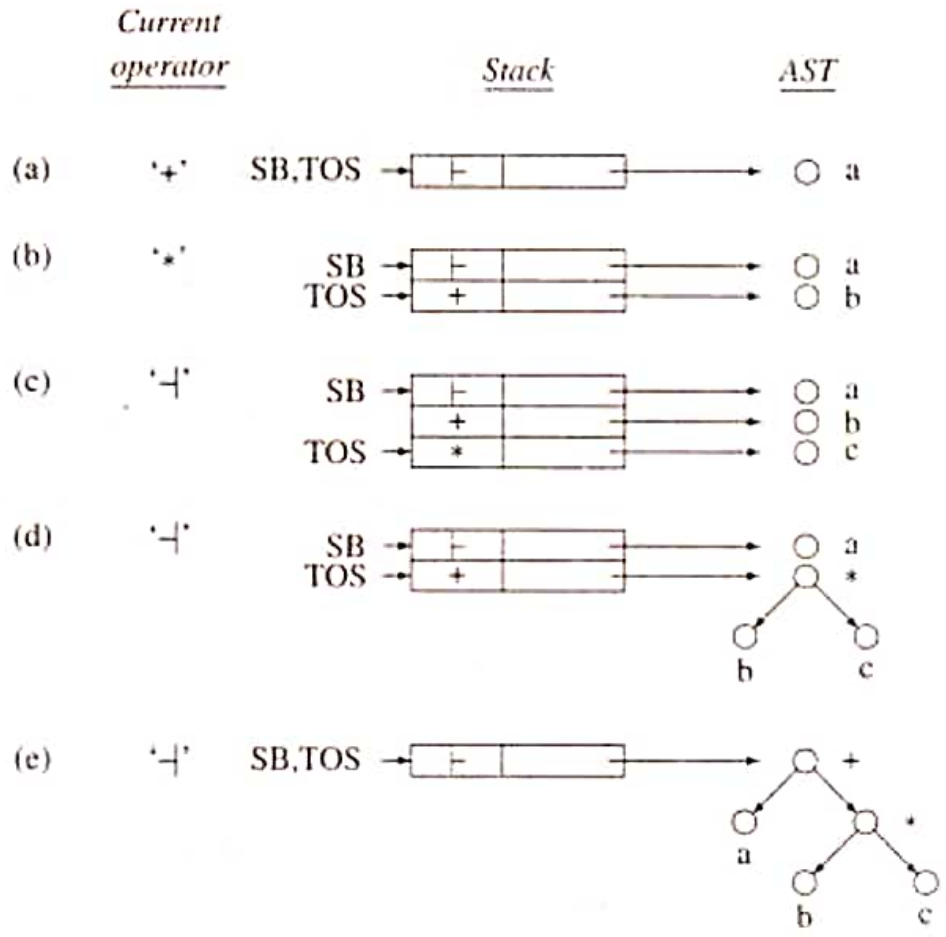


Fig. 3.11 Operator precedence parsing using a stack

LALR parsing

Practical LR parsers are table driven in nature, i.e. the parsing actions are governed by the table entry corresponding to the current state of the parser and the next few source symbols. The parser tables are typically generated using a parser generator. The information in the LR parsing tables and the operation of the LR parsers are described in Dhamdhere(1983) and Aho, Sethi, Ullman (1986). Here, we shall see a simple example containing specification of grammar productions and semantic actions in a YACC like manner. Note that YACC generates a parser which uses the LALR(1) variant of LR parsing.

Example 3.17 Figure 3.12 shows the YACC like specification for an expression parser. The parser builds an AST analogous to the parser of Ex. 3.16. As described in Section 1.5.2, the semantic action specified for a production is executed when a reduction is performed according to that production. The `build_node` routine builds a node

and returns a pointer to it. The node description consists of three parts—node label and pointers to its left and right child nodes. The pointer returned by build_node is remembered as the attribute of the left hand side symbol of a production.

```
%%
E    :  E + T   {$$ = build_node('+', $1, $3)}
     |  Term    {$$ = $1}
     ;
T    :  T * V   {$$ = build_node('*', $1, $3)}
     |  V       {$$ = $1}
     ;
V    :  id      {$$ = build_node($1, nil, nil)}
     ;

%%
build_node(node_label, pointer_1, pointer_2);
{
        /* Build a node and return a pointer to it */
}
```

Fig. 3.12 YACC specification for an expression parser

EXERCISE 3.2

1. Construct an operator precedence matrix for the operators of a grammar for expressions containing arithmetic, relational and boolean operators.
2. Given the following operator grammar

$$\begin{array}{lcl} S & ::= & \vdash A \dashv \\ A & ::= & VaB \mid \varepsilon \\ B & ::= & VaC \\ C & ::= & VbA \\ V & ::= & <id> \end{array}$$

   (a) Construct an operator precedence matrix for the operators of the grammar.
   (b) Give a bottom up parse for a string containing nine <*id*> symbols.

3. An **if** statement may be written with or without an **else** part. Given the following operator grammar for **if**

$$\begin{array}{rcl} <if_stmt> & ::= & \textbf{if} <exp> \textbf{ then} <stmt> \textbf{ else} <stmt> \\ & & \mid \textbf{if} <exp> \textbf{ then} <stmt> \\ <assignment> & ::= & <var> := <exp> \\ <stmt> & ::= & <assignment> \mid <if_stmt> \end{array}$$

   where **if**, **then** and **else** are operators, find whether the operator precedence relations in this grammar are unique.

BIBLIOGRAPHY

Aho, Sethi and Ullman (1986) discuss automatic construction of scanners. Lewis, Rosenkrantz and Stearns (1976), Dhamdhere (1983) and Aho, Sethi and Ullman (1986) discuss parsing techniques in detail.

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.
2. Barrett, W.A. and J.D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.
3. Dhamdhere, D.M. (1983): *Compiler Construction – Principles & Practice*, Macmillan India, New Delhi.
4. Fischer, C.N. and R.J. LeBlanc (1988): *Crafting a Compiler*, Benjamin/Cummings, Menlo Park, California.
5. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.
6. Lewis, P.M., D.J. Rosenkrantz, and R.E. Stearns (1976): *Compiler Design Theory*, Addison-Wesley, Reading.
7. Tremblay, J.P. and P.G. Sorenson (1984): *The Theory and Practice of Compiler Writing*, McGraw-Hill.

CHAPTER 4

Assemblers

4.1 ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

An assembly language is a machine dependent, low level programming language which is specific to a certain computer system (or a family of computer systems). Compared to the machine language of a computer system, it provides three basic features which simplify programming:

1. *Mnemonic operation codes:* Use of mnemonic operation codes (also called *mnemonic opcodes*) for machine instructions eliminates the need to memorize numeric operation codes. It also enables the assembler to provide helpful diagnostics, for example indication of misspelt operation codes.
2. *Symbolic operands:* Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements. The assembler performs memory bindings to these names; the programmer need not know any details of the memory bindings performed by the assembler. This leads to a very important practical advantage during program modification as discussed in Section 4.1.2.
3. *Data declarations:* Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constants into their internal machine representation, for example, conversion of -5 into $(11111010)_2$ or 10.5 into $(41A80000)_{16}$.

Statement format

An assembly language statement has the following format:

[Label] <*Opcode*> <*operand spec*>[,<*operand spec*> ..]

where the notation [..] indicates that the enclosed specification is optional. If a label is specified in a statement, it is associated as a symbolic name with the memory word(s) generated for the statement. <*operand spec*> has the following syntax:

<symbolic name> [+*<displacement>*][(*<index register>*)]

Thus, some possible operand forms are: `AREA`, `AREA+5`, `AREA(4)`, and `AREA+5(4)`. The first specification refers to the memory word with which the name `AREA` is associated. The second specification refers to the memory word 5 words away from the word with the name `AREA`. Here '5' is the *displacement* or *offset* from `AREA`. The third specification implies indexing with index register 4—that is, the operand address is obtained by adding the contents of index register 4 to the address of `AREA`. The last specification is a combination of the previous two specifications.

A simple assembly language

In the first half of the chapter we use a simple assembly language to illustrate features of assembly languages and techniques used in assemblers. In this language, each statement has two operands, the first operand is always a register which can be any one of `AREG`, `BREG`, `CREG` and `DREG`. The second operand refers to a memory word using a symbolic name and an optional displacement. (Note that indexing is not permitted.)

Instruction opcode	*Assembly mnemonic*	*Remarks*
00	`STOP`	Stop execution
01	`ADD`	*First operand is modified*
02	`SUB`	*Condition code is set*
03	`MULT`	
04	`MOVER`	Register ← memory move
05	`MOVEM`	Memory ← register move
06	`COMP`	Sets condition code
07	`BC`	Branch on condition
08	`DIV`	Analogous to SUB
09	`READ`	*First operand is not used*
10	`PRINT`	

Fig. 4.1 Mnemonic operation codes

Figure 4.1 lists the mnemonic opcodes for machine instructions. The `MOVE` instructions move a value between a memory word and a register. In the MOVER instruction the second operand is the source operand and the first operand is the target operand. Converse is true for the MOVEM instruction. All arithmetic is performed in a register (i.e. the result replaces the contents of a register) and sets a *condition code*. A comparison instruction sets a condition code analogous to a subtract instruction without affecting the values of its operands. The condition code can be tested by a Branch on Condition (`BC`) instruction. The assembly statement corresponding to it has the format

`BC` *<condition code spec>*, *<memory address>*

It transfers control to the memory word with the address <*memory address*> if the current value of condition code matches <*condition code spec*>. For simplicity, we assume <*condition code spec*> to be a character string with obvious meaning, e.g. GT, EQ, etc. A BC statement with the condition code spec ANY implies unconditional transfer of control. In a machine language program, we show all addresses and constants in decimal rather than in octal or hexadecimal.

Figure 4.2 shows the machine instructions format. The opcode, register operand and memory operand occupy 2, 1 and 3 digits, respectively. The sign is not a part of the instruction. The condition code specified in a BC statement is encoded into the first operand position using the codes 1-6 for the specifications LT, LE, EQ, GT, GE and ANY, respectively. Figure 4.3 shows an assembly language program and an equivalent machine language program.

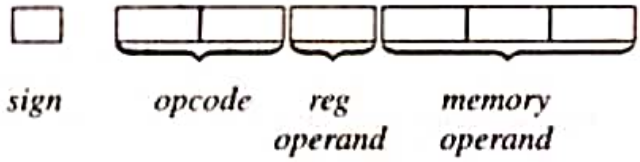


Fig. 4.2 Instruction format

```
        START   101
        READ    N                   101)  + 09 0 113
        MOVER   BREG, ONE           102)  + 04 2 115
        MOVEM   BREG, TERM          103)  + 05 2 116
AGAIN   MULT    BREG, TERM          104)  + 03 2 116
        MOVER   CREG, TERM          105)  + 04 3 116
        ADD     CREG, ONE           106)  + 01 3 115
        MOVEM   CREG, TERM          107)  + 05 3 116
        COMP    CREG, N             108)  + 06 3 113
        BC      LE, AGAIN           109)  + 07 2 104
        MOVEM   BREG, RESULT        110)  + 05 2 114
        PRINT   RESULT              111)  + 10 0 114
        STOP                        112)  + 00 0 000
N       DS      1                   113)
RESULT  DS      1                   114)
ONE     DC      '1'                 115)  + 00 0 001
TERM    DS      1                   116)
        END
```

Fig. 4.3 An assembly and equivalent machine language program

4.1.1 Assembly Language Statements

An assembly program contains three kinds of statements:

1. Imperative statements

2. Declaration statements
3. Assembler directives.

Imperative statements

An imperative statement indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.

Declaration statements

The syntax of declaration statements is as follows:

```
[Label]   DS        <constant>
[Label]   DC        '<value>'
```

The DS (short for *declare storage*) statement reserves areas of memory and associates names with them. Consider the following DS statements:

```
A         DS        1
G         DS        200
```

The first statement reserves a memory area of 1 word and associates the name A with it. The second statement reserves a block of 200 memory words. The name G is associated with the first word of the block. Other words in the block can be accessed through offsets from G, e.g. G+5 is the sixth word of the memory block, etc.

The DC (short for *declare constant*) statement constructs memory words containing constants. The statement

```
ONE       DC        '1'
```

associates the name ONE with a memory word containing the value '1'. The programmer can declare constants in different forms—decimal, binary, hexadecimal, etc. The assembler converts them to the appropriate internal form.

Use of constants

Contrary to the name 'declare constant', the DC statement does not really implement constants, it merely initializes memory words to given values. These values are not protected by the assembler; they may be changed by moving a new value into the memory word. For example, in Fig. 4.3 the value of ONE can be changed by executing an instruction MOVEM BREG, ONE.

An assembly program can use constants in the sense implemented in an HLL in two ways—as immediate operands, and as literals. Immediate operands can be used in an assembly statement only if the architecture of the target machine includes the necessary features. In such a machine, the assembly statement

```
ADD        AREG,5
```

is translated into an instruction with two operands—AREG and the value '5' as an immediate operand. Note that our simple assembly language does not support this feature, whereas the assembly language of Intel 8086 supports it (see Section 4.5).

```
                                ADD     AREG, FIVE
ADD  AREG, ='5'     ⇒           -- --
                         FIVE   DC      '5'
      (a)                           (b)
```

Fig. 4.4 Use of literals in an assembly program

A *literal* is an operand with the syntax ='<*value*>'. It differs from a constant because its location cannot be specified in the assembly program. This helps to ensure that its value is not changed during execution of a program. It differs from an immediate operand because no architectural provision is needed to support its use. An assembler handles a literal by mapping its use into other features of the assembly language. Figure 4.4(a) shows use of a literal ='5'. Figure 4.4(b) shows an equivalent arrangement using a DC statement `FIVE  DC  '5'`. When the assembler encounters the use of a literal in the operand field of a statement, it handles the literal using an arrangement similar to that shown in Fig. 4.4(b)—it allocates a memory word to contain the value of the literal, and replaces the use of the literal in a statement by an operand expression referring to this word. The value of the literal is protected by the fact that the name and address of this word is not know to the assembly language programmer.

Assembler directives

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some assembler directives are described in the following.

START <*constant*>

This directive indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <*constant*>.

END [<*operand spec*>]

This directive indicates the end of the source program. The optional <*operand spec*> indicates the address of the instruction where the execution of the program should begin. (By default, execution begins with the first instruction of the assembled program.)

4.1.2 Advantages of Assembly Language

The primary advantages of assembly language programming vis-a-vis machine language programming arise from the use of symbolic operand specifications. Consider

the machine and assembly language statements of Fig. 4.3 once again. The programs presently compute N!. Figure 4.5 shows a changed program to compute $\frac{1}{2} \times$ N!, where rectangular boxes are used to highlight changes in the program. One statement has been inserted before the PRINT statement to implement division by 2. In the machine language program, this leads to changes in addresses of constants and reserved memory areas. Because of this, addresses used in most instructions of the program had to change. Such changes are not needed in the assembly program since operand specifications are symbolic in nature.

```
         START   101
         READ    N                  101)  + 09 0 114
         MOVER   BREG, ONE          102)  + 04 2 116
         MOVEM   BREG, TERM         103)  + 05 2 117
AGAIN    MULT    BREG, TERM         104)  + 03 2 117
         MOVER   CREG, TERM         105)  + 04 3 117
         ADD     CREG, ONE          106)  + 01 3 116
         MOVEM   CREG, TERM         107)  + 05 3 117
         COMP    CREG, N            108)  + 06 3 114
         BC      LE, AGAIN          109)  + 07 2 104
         DIV     BREG, TWO          110)  + 08 2 118
         MOVEM   BREG, RESULT       111)  + 05 2 115
         PRINT   RESULT             112)  + 10 0 115
         STOP                       113)  + 00 0 000
N        DS      1                  114)
RESULT   DS      1                  115)
ONE      DC      '1'                116)  + 00 0 001
TERM     DS      1                  117)
TWO      DC      '2'                118)  + 00 0 001
         END
```

Fig. 4.5 Modified assembly and machine language programs

Assembly language programming holds an edge over HLL programming in situations where it is necessary or desirable to use specific architectural features of a computer—for example, special instructions supported by the CPU.

4.2 A SIMPLE ASSEMBLY SCHEME

The fundamental translation model is motivated by Definition 1.2. In this section we use this model to develop preliminary ideas on the design of an assembler. We will use these ideas in Sections 4.4 and 4.5.

Design specification of an assembler

We use a four step approach to develop a design specification for an assembler:

1. Identify the information necessary to perform a task.

2. Design a suitable data structure to record the information.
3. Determine the processing necessary to obtain and maintain the information.
4. Determine the processing necessary to perform the task.

The fundamental information requirements arise in the synthesis phase of an assembler. Hence it is best to begin by considering the information requirements of the synthesis tasks. We then consider how to make this information available, i.e. whether it should be collected during analysis or derived during synthesis.

Synthesis phase

Consider the assembly statement

```
MOVER     BREG, ONE
```

in Fig. 4.3. We must have the following information to synthesize the machine instruction corresponding to this statement:

1. Address of the memory word with which name `ONE` is associated,
2. Machine operation code corresponding to the mnemonic `MOVER`.

The first item of information depends on the source program. Hence it must be made available by the analysis phase. The second item of information does not depend on the source program, it merely depends on the assembly language. Hence the synthesis phase can determine this information for itself.

Based on the above discussion, we consider the use of two data structures during the synthesis phase:

1. Symbol table
2. Mnemonics table.

Each entry of the symbol table has two primary fields—*name* and *address*. The table is built by the analysis phase. An entry in the mnemonics table has two primary fields—*mnemonic* and *opcode*. The synthesis phase uses these tables to obtain the machine address with which a name is associated, and the machine opcode corresponding to a mnemonic, respectively. Hence the tables have to be searched with the symbol name and the mnemonic as keys.

Analysis phase

The primary function performed by the analysis phase is the building of the symbol table. For this purpose it must determine the addresses with which the symbolic names used in a program are associated. It is possible to determine some addresses directly, e.g. the address of the first instruction in the program, however others must be inferred. Consider the assembly program of Fig. 4.3. To determine the address of

N, we must fix the addresses of all program elements preceding it. This function is called *memory allocation*.

To implement memory allocation a data structure called *location counter* (LC) is introduced. The location counter is always made to contain the address of the next memory word in the target program. It is initialized to the constant specified in the START statement. Whenever the analysis phase sees a label in an assembly statement, it enters the label and the contents of LC in a new entry of the symbol table. It then finds the number of memory words required by the assembly statement and updates the LC contents. (Hence the word 'counter' in 'location counter'.) This ensures that LC points to the next memory word in the target program even when machine instructions have different lengths and DS/DC statements reserve different amounts of memory. To update the contents of LC, analysis phase needs to know lengths of different instructions. This information simply depends on the assembly language, hence the mnemonics table can be extended to include this information in a new field called *length*. We refer to the processing involved in maintaining the location counter as *LC processing*.

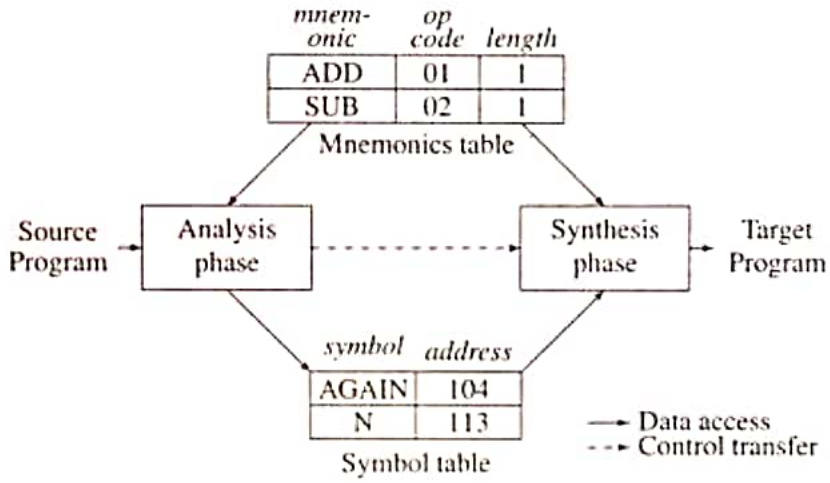


Fig. 4.6 Data structures of the assembler

Figure 4.6 illustrates the use of the data structures by the analysis and synthesis phases. Note that the Mnemonics table is a fixed table which is merely accessed by the analysis and synthesis phases, while the Symbol table is constructed during analysis and used during synthesis. The tasks performed by the analysis and synthesis phases are as follows:

Analysis phase

1. Isolate the label, mnemonic opcode and operand fields of a statement.

2. If a label is present, enter the pair (*symbol*, <*LC contents*>) in a new entry of symbol table.
3. Check validity of the mnemonic opcode through a look-up in the Mnemonics table.
4. Perform LC processing, i.e. update the value contained in LC by considering the opcode

2. If a label is present, enter the pair (*symbol*, <*LC contents*>) in a new entry of symbol table.
3. Check validity of the mnemonic opcode through a look-up in the Mnemonics table.
4. Perform LC processing, i.e. update the value contained in LC by considering the opcode and operands of the statement.

Synthesis phase

1. Obtain the machine opcode corresponding to the mnemonic from the Mnemonics table.
2. Obtain address of a memory operand from the Symbol table.
3. Synthesize a machine instruction or the machine form of a constant, as the case may be.

4.3 PASS STRUCTURE OF ASSEMBLERS

In Section 1.3 we have defined a pass of a language processor as one complete scan of the source program, or its equivalent representation (see Definition 1.4). We discuss two pass and single pass assembly schemes in this section.

Two pass translation

Two pass translation of an assembly language program can handle forward references easily. LC processing is performed in the first pass and symbols defined in the program are entered into the symbol table. The second pass synthesizes the target form using the address information found in the symbol table. In effect, the first pass performs analysis of the source program while the second pass performs synthesis of the target program. The first pass constructs an intermediate representation (IR) of the source program for use by the second pass (see Fig. 4.7). This representation consists of two main components—data structures, e.g. the symbol table, and a processed form of the source program. The latter component is called *intermediate code* (IC).

Single pass translation

LC processing and construction of the symbol table proceed as in two pass translation. The problem of forward references is tackled using a process called *backpatching*. The operand field of an instruction containing a forward reference is left blank initially. The address of the forward referenced symbol is put into this field when its definition is encountered. In the program of Fig. 4.3, the instruction corresponding to the statement

```
MOVER     BREG, ONE
```

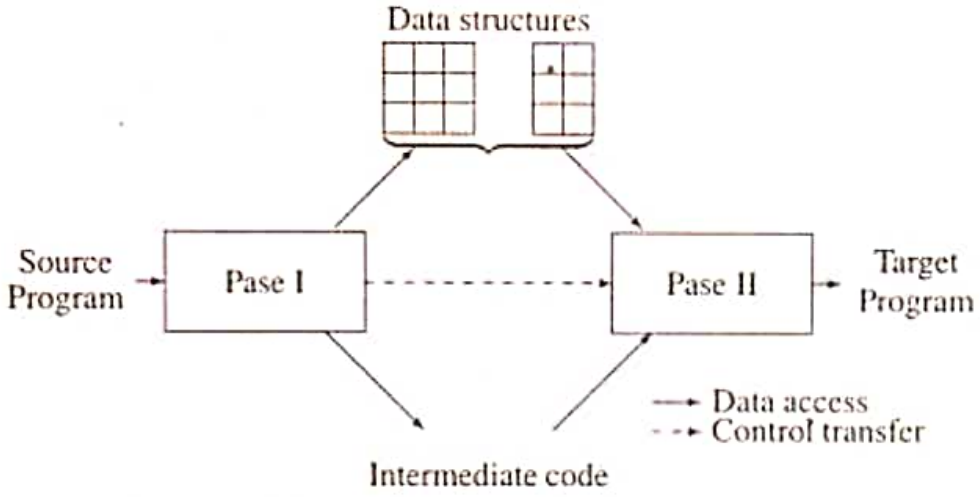


Fig. 4.7 Overview of two pass assembly

can be only partially synthesized since ONE is a forward reference. Hence the instruction opcode and address of BREG will be assembled to reside in location 101. The need for inserting the second operand's address at a later stage can be indicated by adding an entry to the Table of Incomplete Instructions (TII). This entry is a pair (<*instruction address*>, <*symbol*>), e.g. (101, ONE) in this case.

By the time the END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program and TII would contain information describing all forward references. The assembler can now process each entry in TII to complete the concerned instruction. For example, the entry (101, ONE) would be processed by obtaining the address of ONE from symbol table and inserting it in the operand address field of the instruction with assembled address 101. Alternatively, entries in TII can be processed in an incremental manner. Thus, when definition of some symbol *symb* is encountered, all forward references to *symb* can be processed.

4.4 DESIGN OF A TWO PASS ASSEMBLER

Tasks performed by the passes of a two pass assembler are as follows:

Pass I
1. Separate the symbol, mnemonic opcode and operand fields.
2. Build the symbol table.
3. Perform LC processing.
4. Construct intermediate representation.

Pass II Synthesize the target program.

Pass I performs analysis of the source program and synthesis of the intermediate representation while Pass II processes the intermediate representation to synthesize the

target program. The design details of assembler passes are discussed after introducing advanced assembler directives and their influence on LC processing.

4.4.1 Advanced Assembler Directives

ORIGIN

The syntax of this directive is

ORIGIN <*address spec*>

where <*address spec*> is an <*operand spec*> or <*constant*>. This directive indicates that LC should be set to the address given by <*address spec*>. The ORIGIN statement is useful when the target program does not consist of consecutive memory words: The ability to use an <*operand spec*> in the ORIGIN statement provides the ability to perform LC processing in a *relative* rather than *absolute* manner. Example 4.1 illustrates the differences between the two.

Example 4.1 Statement number 18 of Fig. 4.8(a), viz. ORIGIN LOOP+2, sets LC to the value 204, since the symbol LOOP is associated with the address 202. The next statement, viz.

```
MULT      CREG, B
```

is therefore given the address 204. The statement ORIGIN LAST+1 sets LC to address 217. Note that an equivalent effect could have been achieved by using the statements ORIGIN 202 and ORIGIN 217 at these two places in the program, however the absolute addresses used in these statements would need to be changed if the address specification in the START statement is changed.

EQU

The EQU statement has the syntax

<*symbol*> EQU <*address spec*>

where <*address spec*> is an <*operand spec*> or <*constant*>.

The EQU statement defines the symbol to represent <*address spec*>. This differs from the DC/DS statement as no LC processing is implied. Thus EQU simply associates the name <*symbol*> with <*address spec*>.

Example 4.2 Statement 22 of Fig. 4.8(a), viz. BACK EQU LOOP introduces the symbol BACK to represent the operand LOOP. This is how the 16th statement, viz.

```
BC        LT, BACK
```

is assembled as '+ 07 1 202'.

```
 1          START   200
 2          MOVER   AREG, ='5'        200)  +04 1 211
 3          MOVEM   AREG, A           201)  +05 1 217
 4   LOOP   MOVER   AREG, A           202)  +04 1 217
 5          MOVER   CREG, B           203)  +05 3 218
 6          ADD     CREG, ='1'        204)  +01 3 212
 7          ...

12          BC      ANY, NEXT         210)  +07 6 214
13          LTORG
                    ='5'              211)  +00 0 005
                    ='1'              212)  +00 0 001
14          ...
15   NEXT   SUB     AREG, ='1'        214)  +02 1 219
16          BC      LT, BACK          215)  +07 1 202
17   LAST   STOP                      216)  +00 0 000
18          ORIGIN  LOOP+2
19          MULT    CREG, B           204)  +03 3 218
20          ORIGIN  LAST+1
21   A      DS      1                 217)
22   BACK   EQU     LOOP
23   B      DS      1                 218)
24          END
25                  ='1'              219)  +00 0 001
```

Fig. 4.8 An assembly program illustrating ORIGIN

LTORG

Fig. 4.4 has shown how literals can be handled in two steps. First, the literal is treated as if it is a <*value*> in a DC statement, i.e. a memory word containing the value of the literal is formed. Second, this memory word is used as the operand in place of the literal. Where should the assembler place the word corresponding to the literal? Obviously, it should be placed such that control never reaches it during the execution of a program. The LTORG statement permits a programmer to specify where literals should be placed. By default, assembler places the literals after the END statement.

At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of a *literal pool*. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.

Example 4.3 In Fig. 4.8, the literals ='5' and ='1' are added to the literal pool in statements 2 and 6, respectively. the first LTORG statement (statement number 13) allocates the addresses 211 and 212 to the values '5' and '1'. A new literal pool is now started. The value '1' is put into this pool in statement 15. This value is allocated the address 219 while processing the END statement. The literal ='1' used in statement 15 therefore refers to location 219 of the second pool of literals rather than location 212 of the first pool. Thus, all references to literals are forward references by definition.

The LTORG directive has very little relevance for the simple assembly language we have assumed so far. The need to allocate literals at intermediate points in the program rather than at the end is critically felt in a computer using a base displacement mode of addressing, e.g. computers of the IBM 360/370 family.

EXERCISE 4.4.1

1. An assembly program contains the statement

```
X        EQU        Y+25
```

   Indicate how the EQU statement can be processed if

   (a) Y is a back reference,

   (b) Y is a forward reference.

2. Can the operand expression in an ORIGIN statement contain forward references? If so, outline how the statement can be processed in a two pass assembly scheme.

4.4.2 Pass I of the Assembler

Pass I uses the following data structures:

OPTAB	A table of mnemonic opcodes and related information
SYMTAB	Symbol table
LITTAB	A table of literals used in the program

Figure 4.9 illustrates sample contents of these tables while processing the program of Fig. 4.8. OPTAB contains the fields *mnemonic opcode*, *class* and *mnemonic info*. The *class* field indicates whether the opcode corresponds to an imperative statement (IS), a declaration statement (DL) or an assembler directive (AD). If an imperative, the *mnemonic info* field contains the pair (*machine opcode*, *instruction length*), else it contains the id of a routine to handle the declaration or directive statement. A SYMTAB entry contains the fields *address* and *length*. A LITTAB entry contains the fields *literal* and *address*.

Processing of an assembly statement begins with the processing of its label field. If it contains a symbol, the symbol and the value in LC is copied into a new entry of SYMTAB. Thereafter, the functioning of Pass I centers around the interpretation of the OPTAB entry for the mnemonic. The *class* field of the entry is examined to determine whether the mnemonic belongs to the class of imperative, declaration or assembler directive statements. In the case of an imperative statement, the length of the machine instruction is simply added to the LC. The length is also entered in the SYMTAB entry of the symbol (if any) defined in the statement. This completes the processing of the statement.

For a declaration or assembler directive statement, the routine mentioned in the *mnemonic info* field is called to perform appropriate processing of the statement. For example, in the case of a DS statement, routine R#7 would be called. This routine

mnemonic opcode	*class*	*mnemonic info*
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

OPTAB

symbol	*address*	*length*
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

SYMTAB

	literal	*address*
1	='5'	
2	='1'	
3	='1'	

LITTAB

literal no
#1
#3
–

POOLTAB

Fig. 4.9 Data structures of assembler Pass I

processes the operand field of the statement to determine the amount of memory required by this statement and appropriately updates the LC and the SYMTAB entry of the symbol (if any) defined in the statement. Similarly, for an assembler directive the called routine would perform appropriate processing, possibly affecting the value in LC.

The use of LITTAB needs some explanation. The first pass uses LITTAB to collect all literals used in a program. Awareness of different literal pools is maintained using the auxiliary table POOLTAB. This table contains the literal number of the starting literal of each literal pool. At any stage, the current literal pool is the last pool in LITTAB. On encountering an LTORG statement (or the END statement), literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented. Thus, the literals of the program in Fig. 4.8(a) will be allocated memory in two steps. At the LTORG statement, the first two literals will be allocated the addresses 211 and 212. At the END statement, the third literal will be allocated address 219.

We now present the algorithm for the first pass of the assembler. Intermediate code forms for use in a two pass assembler are discussed in the next section.

Algorithm 4.1 (Assembler First Pass)

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB [1] := 1;
   littab_ptr := 1;
2. While next statement is not an END statement
   (a) If label is present then
      this_label := symbol in label field;
      Enter (*this_label*, *loc_cntr*) in SYMTAB.
   (b) If an LTORG statement then
      (i) Process literals LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [*littab_ptr*−1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.
      (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
      (iii) POOLTAB [*pooltab_ptr*] := *littab_ptr*;
   (c) If a START or ORIGIN statement then
      loc_cntr := value specified in operand field;
   (d) If an EQU statement then
      (i) *this_addr* := value of <*address spec*>;
      (ii) Correct the symtab entry for *this_label* to (*this_label*, *this_addr*).
   (e) If a declaration statement then
      (i) *code* := code of the declaration statement;
      (ii) *size* := size of memory area required by DC/DS.
      (iii) *loc_cntr* := *loc_cntr* + *size*;
      (iv) Generate IC '(DL, *code*) ···'.
   (f) If an imperative statement then
      (i) *code* := machine opcode from OPTAB;
      (ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;
      (iii) If operand is a literal then
         this_literal := literal in operand field;
         LITTAB [*littab_ptr*] := *this_literal*;
         littab_ptr := *littab_ptr* + 1;
      else (i.e. operand is a symbol)
         this_entry := SYMTAB entry number of operand;
         Generate IC '(IS, *code*)(S, *this_entry*)';
3. (Processing of END statement)
   (a) Perform step 2(b).
   (b) Generate IC '(AD,02)'.
   (c) Go to Pass II.

4.4.3 Intermediate Code Forms

In Section 1.3 two criteria for the choice of intermediate code, viz. processing efficiency and memory economy, have been mentioned. In this section we consider some variants of intermediate codes and compare them on the basis of these criteria.

The intermediate code consists of a set of IC units, each IC unit consisting of the following three fields (see Fig. 4.10):

1. Address
2. Representation of the mnemonic opcode
3. Representation of operands.

Address	Opcode	Operands

Fig. 4.10 An IC unit

Variant forms of intermediate codes, specifically the operand and address fields, arise in practice due to the tradeoff between processing efficiency and memory economy. These variants are discussed in separate sections dealing with the representation of imperative statements, and declaration statements and directives, respectively. The information in the mnemonic field is assumed to have the same representation in all the variants.

Mnemonic field

The mnemonic field contains a pair of the form

(*statement class, code*)

where *statement class* can be one of IS, DL and AD standing for imperative statement, declaration statement and assembler directive, respectively. For an imperative statement, *code* is the instruction opcode in the machine language. For declarations and assembler directives, *code* is an ordinal number within the class. Thus, (AD, 01) stands for assembler directive number 1 which is the directive START. Figure 4.11 shows the codes for various declaration statements and assembler directives.

Declaration statements		*Assembler directives*	
DC	01	START	01
DS	02	END	02
		ORIGIN	03
		EQU	04
		LTORG	05

Fig. 4.11 Codes for declaration statements and directives

4.4.4 Intermediate Code for Imperative Statements

We consider two variants of intermediate code which differ in the information contained in their operand fields. For simplicity, the address field is assumed to contain identical information in both variants.

Variant I

The first operand is represented by a single digit number which is a code for a register (1-4 for AREG-DREG) or the condition code itself (1-6 for LT-ANY). The second operand, which is a memory operand, is represented by a pair of the form

(operand class, code)

where *operand class* is one of C, S and L standing for constant, symbol and literal, respectively (see Fig. 4.12). For a constant, the *code* field contains the internal representation of the constant itself. For example, the operand descriptor for the statement START 200 is (C, 200). For a symbol or literal, the *code* field contains the ordinal number of the operand's entry in SYMTAB or LITTAB. Thus entries for a symbol XYZ and a literal ='25' would be of the form (S, 17) and (L, 35) respectively.

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   (S,01)
LOOP    MOVER   AREG, A         (IS,04)   (1)(S,01)
          ⋮                       ⋮
        SUB     AREG, ='1'      (IS,02)   (1)(L,01)
        BC      GT, LOOP        (IS,07)   (4)(S,02)
        STOP                    (IS,00)
A       DS      1               (DL, 02)  (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

Fig. 4.12 Intermediate code - variant I

Note that this method of representing symbolic operands gives rise to one peculiarity. We have so far assumed that an entry is made in SYMTAB only when a symbol occurs in the label field of an assembly statement, e.g. an entry (A, 345, 1) if symbol A is allocated one word at address 345. However, while processing a forward reference

```
MOVER AREG, A
```

it is necessary to enter A in SYMTAB, say in entry number *n*, so that it can be represented by (S, *n*) in IC. At this point, the *address* and *length* fields of A's entry cannot be filled in. This implies that two kinds of entries may exist in SYMTAB at any time—for defined symbols and for forward references. This fact should be noted for use during error detection (see Section 4.4.7).

Variant II

This variant differs from variant I of the intermediate code in that the operand fields of the source statements are selectively replaced by their processed forms (see Fig. 4.13). For declarative statements and assembler directives, processing of the operand fields is essential to support LC processing. Hence these fields contain the processed forms. For imperative statements, the operand field is processed only to identify literal references. Literals are entered in LITTAB, and are represented as (L, *m*) in IC. Symbolic references in the source statement are not processed at all during Pass I.

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   A
LOOP    MOVER   AREG, A         (IS,04)   AREG, A
          ⋮                       ⋮
        SUB     AREG, ='1'      (IS,02)   AREG, (L,01)
        BC      GT, LOOP        (IS,07)   GT, LOOP
        STOP                    (IS,00)
A       DS      1               (DL,02)   (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

Fig. 4.13 Intermediate code - variant II

Comparison of the variants

Variant I of the intermediate code appears to require extra work in Pass I since operand fields are completely processed. However, this processing considerably simplifies the tasks of Pass II—a look at the IC of Fig. 4.12 confirms this. The functions of Pass II are quite trivial. To process the operand field of a declaration statement, we only need to refer to the appropriate table and obtain the operand address. Most declarations do not require any processing, e.g. DC, DS (see Section 4.4.5), and START statements, while some, e.g. LTORG, require marginal processing. The IC is quite compact—it can be as compact as the target code itself if each operand reference like (S, *n*) can be represented in the same number of bits as an operand address in a machine instruction.

Variant II reduces the work of Pass I by transferring the burden of operand processing from Pass I to Pass II of the assembler. The IC is less compact since the memory operand of a typical imperative statement is in the source form itself. On the other hand, by making Pass II to perform more work, the functions and memory requirements of the two passes get better balanced. Figure 4.14 illustrates the advantages of this aspect. Part (a) of Fig. 4.14 shows memory utilization by an assembler using variant I of IC. Some data structures, viz. symbol table, are passed in the memory while IC is presumably written in a file. Since Pass I performs much more processing than Pass II, its code occupies more memory than the code of Pass II. Part

(b) of Fig. 4.14 shows memory utilization when variant II of IC is used. The code sizes of the two passes are now comparable, hence the overall memory requirement of the assembler is lower.

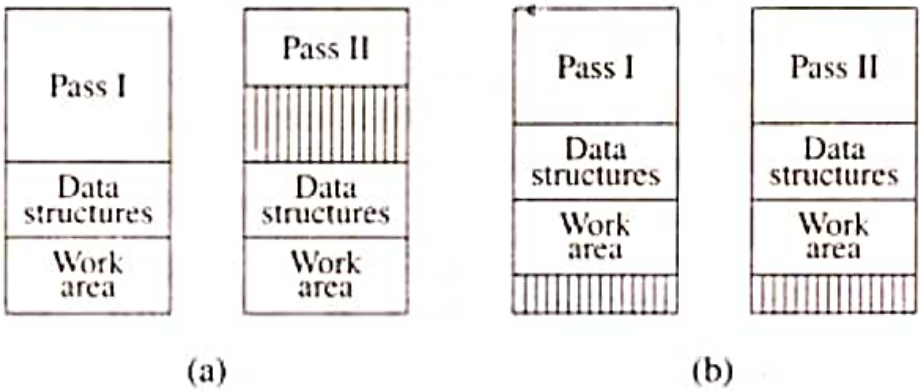


Fig. 4.14 Memory requirements using (a) variant I, (b) variant II

Variant II is particularly well-suited if expressions are permitted in the operand fields of an assembly statement. For example, the statement

```
MOVER      AREG, A+5
```

would appear as

```
(IS,05)    (1) (S,01)+5
```

in variant I of IC. This does not particularly simplify the task of Pass II or save much memory space. In such situations, it would have been preferable not to have processed the operand field at all.

4.4.5 Processing of Declarations and Assembler Directives

The focus of this discussion is on identifying alternative ways of processing declaration statements and assembler directives. In this context, it is useful to consider how far these statements can be processed in Pass I of the assembler. This depends on answers to two related questions:

1. Is it necessary to represent the address of each source statement in IC?
2. Is it necessary to have an explicit representation of DS statements and assembler directives in IC?

Let the answer to the first question be 'yes'. Now consider the following source program fragment and its intermediate code:

```
        START  200          —)  (AD,01)  (C,200)
AREA1   DS     20     ⇒   200)  (DL,02)  (C,20)
SIZE    DC     5           220)  (DL,01)  (C,5)
```

Here, it is redundant to have the representations of the START and DS statements in IC, since the effect of these statements is implied in the fact that the DC statement has the address 220 ! Thus, it is not necessary to have a representation for DS statements and assembler directives in IC if the IC contains an *address* field. If the *address* field of the IC is omitted, a representation for the DS statements and assembler directives becomes essential. Now, Pass II can determine the address for SIZE only after analyzing the intermediate code units for the START and DS statements. The first alternative avoids this processing but requires the existence of the address field. Yet another instance of space–time tradeoff !

DC statement

A DC statement must be represented in IC. The mnemonic field contains the pair (DL,01). The operand field may contain the value of the constant in the source form or in the internal machine representation. No processing advantage exists in either case since conversion of the constant into the machine representation is required anyway. If a DC statement defines many constants, e.g.

```
DC        '5, 3, -7'
```

a series of (DL,01) units can be put in the IC.

START and ORIGIN

These directives set new values into the LC. It is not necessary to retain START and ORIGIN statements in the IC if the IC contains an address field.

LTORG

Pass I checks for the presence of a literal reference in the operand field of every statement. If one exists, it enters the literal in the current literal pool in LITTAB. When an LTORG statement appears in the source program, it assigns memory addresses to the literals in the current pool. These addresses are entered in the *address* field of their LITTAB entries.

After performing this fundamental action, two alternatives exist concerning Pass I processing. Pass I could simply construct an IC unit for the LTORG statement and leave all subsequent processing to Pass II. Values of literals can be inserted in the target program when this IC unit is processed in Pass II. This requires the use of POOLTAB and LITTAB in a manner analogous to Pass I.

Example 4.4 Figure 4.9 shows the LITTAB and POOLTAB for the program of Fig. 4.8 at the end of Pass I. Literals of the first pool are copied into the target program when the IC unit for LTORG is encountered in Pass II. Literals of the second pool are copied into the target program when the IC unit for END is processed.

Alternatively, Pass I could itself copy out the literals of the pool into the IC. This avoids duplication of Pass I actions in Pass II. The IC for a literal can be made

identical to the IC for a DC statement so that no special processing is required in Pass II.

Example 4.5 Figure 4.15 shows the IC for the first half of the program of Fig. 4.8. The literals of the first pool (see Fig. 4.9) are copied out at LTORG statement. Note that the opcode field of the IC units, i.e (DL,01), is same as that for DC statements.

```
        START   200              (AD,01)   (C,200)
        MOVER   AREG, ='5'       (IS,04)   (1)(L,01)
        MOVEM   AREG, A          (IS,05)   (1)(S,01)
LOOP    MOVER   AREG, A          (IS,04)   (1)(S,01)
          :
        BC      ANY, NEXT        (IS,07)   (6)(S,04)
        LTORG                    (DL,01)   (C,5)
                                 (DL,01)   (C,1)
          :
```

Fig. 4.15 Copying of literal values into intermediate code

However, this alternative increases the tasks to be performed by Pass I, consequently increasing its size. This might lead to an unbalanced pass structure for the assembler with the consequences illustrated in Fig. 4.14. Secondly, the literals have to exist in two forms simultaneously, in the LITTAB along with the address information, and also in the intermediate code.

EXERCISE 4.4

1. Given the following source program:

```
        START    100
A       DS       3
L1      MOVER    AREG, B
        ADD      AREG, C
        MOVEM    AREG, D
D       EQU      A+1
L2      PRINT    D

        ORIGIN   A-1
C       DC       '5'
        ORIGIN   L2+1
        STOP
B       DC       '19'
        END      L1
```

   (a) Show the contents of the symbol table at the end of Pass I.
   (b) Explain the significance of EQU and ORIGIN statements in the program and explain how they are processed by the assembler.
   (c) Show the Intermediate code generated for the program.

4.4.6 Pass II of the Assembler

Algorithm 4.2 is the algorithm for assembler Pass II. Minor changes may be needed to suit the IC being used. It has been assumed that the target code is to be assembled in the area named *code_area*.

Algorithm 4.2 (Assembler Second Pass)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   loc_cntr := 0;
2. While next statement is not an END statement
   (a) Clear *machine_code_buffer*;
   (b) If an LTORG statement
      (i) Process literals in LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [POOLTAB [*pooltab_ptr*+1]]−1 similar to processing of constants in a DC statement, i.e. assemble the literals in *machine_code_buffer*.
      (ii) *size* := size of memory area required for literals;
      (iii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (c) If a START or ORIGIN statement then
      (i) *loc_cntr* := value specified in operand field;
      (ii) *size* := 0;
   (d) If a declaration statement
      (i) If a DC statement then
         Assemble the constant in *machine_code_buffer*.
      (ii) *size* := size of memory area required by DC/DS;
   (e) If an imperative statement
      (i) Get operand address from SYMTAB or LITTAB.
      (ii) Assemble instruction in *machine_code_buffer*.
      (iii) *size* := size of instruction;
   (f) If *size* $\neq$ 0 then
      (i) Move contents of *machine_code_buffer* to the address *code_area_address* + *loc_cntr*;
      (ii) *loc_cntr* := *loc_cntr* + *size*;
3. (Processing of END statement)
   (a) Perform steps 2(b) and 2(f).
   (b) Write *code_area* into output file.

Output interface of the assembler

It has been assumed that the assembler produces a target program which is the machine language of the target computer. This is rarely (if ever !) the case. The assembler produces an *object module* in the format required by a linkage editor or loader. The information contained in object modules is discussed in Chapter 7.

4.4.7 Listing and Error Reporting

Design of an error indication scheme involves some decisions which influence the effectiveness of error reporting and the speed and memory requirements of the assembler. The basic decision is whether to produce program listing and error reports in Pass I or delay these actions until Pass II. Producing the listing in the first pass has the advantage that the source program need not be preserved till Pass II. This conserves memory and avoids some amount of duplicate processing.

This design decision also has very important implications from a programmer's viewpoint. A listing produced in Pass I can report only certain errors in the most relevant place, that is, against the source statement itself. Examples of such errors are syntax errors like missing commas or parentheses and semantic errors like duplicate definitions of symbols. Other errors like references to undefined variables can only be reported at the end of the source program (see Fig. 4.16). The target code can be printed later in Pass II, however it is difficult to locate the target code corresponding to a source statement and vice versa. All these factors make debugging difficult.

```
Sr. No.        Statement              Address
 001          START  200
 002          MOVER  AREG, A          200
 003            :
 009          MVER   BREG, A          207
        ** error ** Invalid opcode
 010          ADD    BREG, B          208
 014    A     DS     1                209
 015            :
 021    A     DC     '5'              227
        ** error ** Duplicate definition of symbol A
 022            :
 035          END
        ** error ** Undefined symbol B in statement 10
```

Fig. 4.16 Error reporting in pass I

For effective error reporting, it is necessary to report all errors against the erro-

neous statement itself. This can be achieved by delaying program listing and error reporting till Pass II. Now the error reports as well as the target code can be printed against each source statement (see Ex. 4.6).

Example 4.6 Figure 4.16 illustrates error reporting in Pass I. Detection of errors in statements 9 and 21 is straightforward. In statement 9, the opcode is known to be invalid because it does not match with any mnemonic in OPTAB. In statement 21, A is known to be a duplicate definition because an entry for A already exists in the symbol table. Use of the undefined symbol B is harder to detect because at the end of Pass I we have no record that a forward reference to B exists in statement 10. This problem can be resolved by making an entry for B in the symbol table with an indication that a forward reference to B exists in statement 10. All such entries would be processed at the end of Pass I to check if a definition of the symbol has been encountered. If not, the symbol table entry contains sufficient information for error reporting. Note that the target instructions cannot be printed because they have not yet been generated. The memory address is printed against each statement in a weak attempt to provide a cross-reference between source statements and target instructions.

Example 4.7 Figure 4.17 illustrates error reporting performed in Pass II. Indication of errors in statements 9 and 21 is as easy as in Ex. 4.6. Indication of the error in statement 10 is equally easy—the symbol table is searched for an entry of B and an error is reported when no matching entry is found. Note that target program instructions appear against the source statements to which they belong.

```
Sr. No.        Statement              Address   Instruction

001          START   200
002          MOVER   AREG, A          200       + 04 1 209
               ⋮
003
009          MVER    BREG, A          207       + -- 2 209
         ** error ** Invalid opcode
010          ADD     BREG, B          208       + 01 2 ---
         ** error ** Undefined symbol B in operand field
014      A   DS      1                209
               ⋮
015
021      A   DC      '5'              227       + 00 0 005
         ** error ** Duplicate definition of symbol A
               ⋮
022
035          END
```

Fig. 4.17 Error reporting in pass II

EXERCISE 4.4.7

1. A two pass assembler performs program listing and error reporting in Pass II using the following strategy: Errors detected in Pass I are stored in an error table. These are reported along with Pass II errors while producing the program listing.

   (a) Design the error table for use by Pass I. What is its entry format? What is the table organization?

   (b) Let the error messages (e.g. `DUPLICATE LABEL...`) be stored in an error message table. Comment on the organization of this table.

   (*Note:* Readers may refer to Dhamdhere (1983) for some interesting error reporting strategies.)

4.4.8 Some Organizational Issues

We discuss some organizational issues in assembler design, like the placement and access of tables and IC, with respect to the schematic shown in Fig. 4.18.

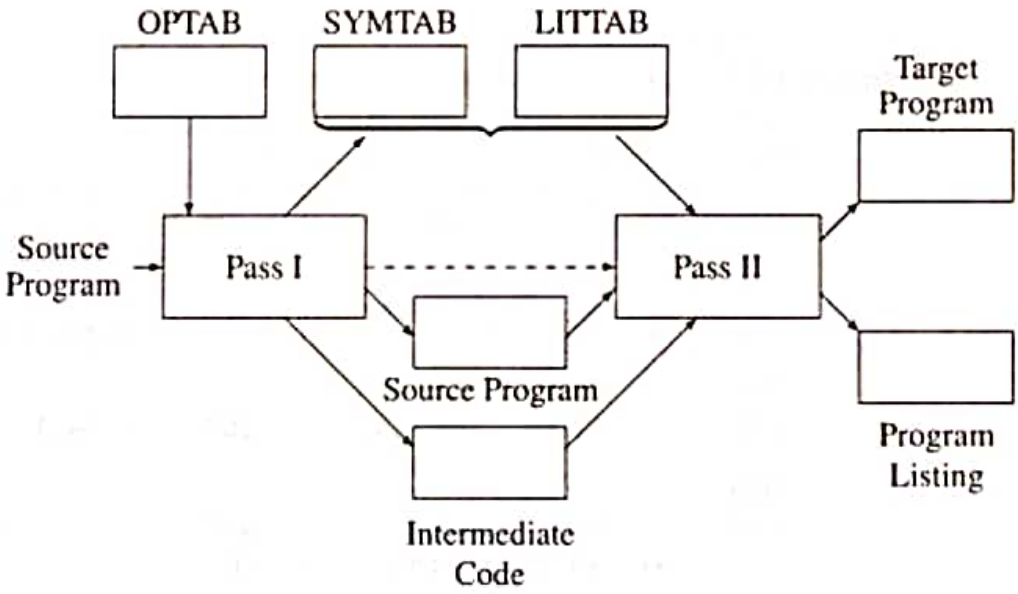


Fig. 4.18 Data structures and files in a two pass assembler

Tables

For efficiency reasons SYMTAB must remain in main memory throughout Passes I and II of the assembler. LITTAB is not accessed as frequently as SYMTAB, however it may be accessed sufficiently frequently to justify its presence in the memory. If memory is at a premium, it is possible to hold only part of LITTAB in the memory because only the literals of the current pool need to be accessible at any time. For obvious reasons, no such partitioning is feasible for SYMTAB. OPTAB should be in memory during Pass I.

Source program and intermediate code

The source program would be read by Pass I on a statement by statement basis. After processing, a source statement can be written into a file for subsequent use in Pass II. The IC generated for it would also be written into another file. The target code and the program listings can be written out as separate files by Pass II. Since all these files are sequential in nature, it is beneficial to use appropriate blocking and buffering of records.

EXERCISE 4.4.8

1. Develop complete program specifications for the passes of a two pass assembler indicating
   (a) Tables for internal use of the passes
   (b) Tables to be shared between passes
   (c) Inputs (files and tables) for every pass
   (d) Outputs (files and tables) of every pass.

   You must clearly specify why certain information is in the form of tables in main memory while other information is in the form of files.
2. Recommend appropriate organizations for the tables and files used in the two pass assembler of problem 1.

4.5 A SINGLE PASS ASSEMBLER FOR IBM PC

We shall discuss a single pass assembler for the intel 8088 processor used in the IBM PC. The discussion focuses on the design features for handling the forward reference problem in an environment using segment-based addressing.

4.5.1 The architecture of Intel 8088

The intel 8088 microprocessor supports 8 and 16 bit arithmetic, and also provides special instructions for string manipulation. The CPU contains the following features (see Fig. 4.19):

- Data registers AX, BX, CX and DX
- Index registers SI and DI
- Stack pointer registers BP and SP
- Segment registers Code, Stack, Data and Extra.

Each data register is 16 bits in size, split into the upper and lower halves. Either half can be used for 8 bit arithmetic, while the two halves together constitute the data register for 16 bit arithmetic. The architecture supports stacks for storing subroutine and interrupt return addresses, parameters and other data. The index registers SI and DI are used to index the source and destination addresses in string manipulation instructions. They are provided with the auto-increment and auto-decrement facility.

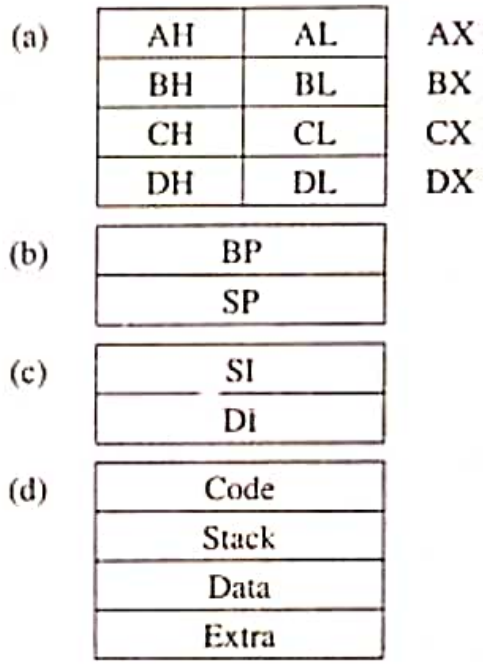


Fig. 4.19 (a) Data, (b) Base, (c) Index and (d) Segment registers

Two stack pointer registers called SP and BP are provided to address the stack. SP points into the stack implicitly used by the architecture to store subroutine and interrupt return addresses. BP can be used by the programmer in any desired manner. Push and Pop instructions are provided for this purpose.

The Intel 8088 provides addressing capability for 1 MB of primary memory. The memory is used to store three components of a program, program code, data and stack. The Code, Stack and Data segment registers are used to contain the start addresses of these three components. The Extra segment register points to another memory area which can be used to store data. To address a memory location, an instruction designates a segment register and provides a 16 bit logical address. The address contained in the segment register is extended by adding four lower order zeroes to yield the segment base address. The logical address is now added to it to obtain a 20 bit memory address. The size of each segment is thus limited to 2^{16}, i.e. 64K, bytes. A large program may contain many segments, only four of which can be addressed at any given time. Segment-based addressing facilitates easy relocation of programs. If the memory area allocated to a program is changed, it is sufficient to change the addresses loaded in the segment registers. The memory addresses generated would now automatically lie in the new memory area occupied by the program.

The 8088 architecture provides 24 addressing modes. These are summarized in Fig. 4.20. In the immediate addressing mode, the instruction itself contains the data that is to participate in the instruction. This data can be 8 or 16 bits in length. In the direct addressing mode, the instruction contains a 16 bit number which is taken to be a displacement from the segment base contained in a segment register. The segment

Addressing mode	*Example*	*Remarks*
Immediate	`MOV SUM, 1234H`	Data = 1234H
Register	`MOV SUM, AX`	AX contains the data
Direct	`MOV SUM, [1234H]`	Data disp. = 1234H
Register indirect	`MOV SUM, [BX]`	Data disp. = (BX)
Register indirect	`MOV SUM, CS: [BX]`	Segment override : Segment base = (CS) Data disp. = (BX)
Based	`MOV SUM, 12H [BX]`	Data disp. = 12H+(BX)
Indexed	`MOV SUM, 34H [SI]`	Data disp. = 34H+(SI)
Based & indexed	`MOV SUM, 56H [SI] [BX]`	Data disp. = 56H + (SI) + (BX)

Fig. 4.20 Addressing modes of 8088 ('(..)' implies 'contents of')

register may be explicitly indicated in a prefix of the instruction, else a default segment register is used. In the indexed mode, contents of the index register indicated in the instruction (SI or DI) are added to the 8 or 16 bit displacement contained in the instruction. The result is taken to be the displacement from the segment base of the data segment. In the based mode, contents of the base register (BP or BX) are added to the displacement. The result is taken to be the displacement from the data segment base unless BP is specified, in which case it is taken as a displacement from the stack segment base. The based-and-indexed-with-displacement mode combines the effect of the based and indexed modes.

4.5.2 Intel 8088 Instructions

Arithmetic instructions

The operands can be in one of the four 16 bit registers, or in a memory location designated by one of the 24 addressing modes. Three instruction formats shown in Fig. 4.21 are supported. The *mod* and *r/m* fields specify the first operand, which can be in a register or in memory, while the *reg* field describes the second operand, which is always a register. The instruction opcode indicates which instruction format is applicable. The direction field (d) in the instruction indicates which operand is the destination operand in the instruction. If $d = 0$, the register/memory operand is the destination, else the register operand indicated by *reg* is the destination. The *width* field (w) indicates whether 8 or 16 bit arithmetic is to be used. The conventions for determining the operands are described in Fig. 4.21.

Figure 4.22 contains some sample instructions. Note that in the first statement, AL could be encoded into the first or the second operand of the instruction. In the second statement, however, it has to be encoded into second operand since the first operand has to be `12H [SI]`. Here, *mod* = 01 since only one byte of displacement is

(a) *Register/Memory to Register*

opcode d w	*mod reg r/m*

(b) *Immediate to Register/Memory*

opcode d w	*mod op r/m*	*data*	*data*

(c) *Immediate to Accumulator*

opcode w	*data*	*data*

r/m	*mod* = 00	*mod* = 01	*mod* = 10	*mod* = 11	
				w=0	*w*=1
000	(BX) + (SI)	(BX) + (SI) + d8	Note 2	AL	AX
001	(BX) + (DI)	(BX) + (DI) + d8	Note 2	CL	CX
010	(BP) + (SI)	(BP) + (SI) + d8	Note 2	DL	DX
011	(BP) + (DI)	(BP) + (DI) + d8	Note 2	BL	BX
100	(SI)	(SI) + d8	Note 2	AH	SP
101	(DI)	(DI) + d8	Note 2	CH	BP
110	Note 1	(BP) + d8	Note 2	DH	SI
111	(BX)	(BX) + d8	Note 2	BH	DI

Note 1 : (BP) + DISP for indirect addressing, d16 for direct
Note 2 : Same as previous column, except d16 instead of d8

	Register	
reg	*8-bit* (*w*=0)	*16-bit* (*w*=1)
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Fig. 4.21 Instruction formats of Intel 8088

Assembly statement	*Opcode d w*	*mod reg r/m*	*data/ displacement*
`ADD AL, BL`	000000 0 0	11 011 000	
`ADD AL, 12H [SI]`	000000 1 0	01 000 100	00010010
`ADD AX, 3456H`	100000 0 1	11 000 000	01010110 00110100
`ADD AX, 3456H`	000001 0 1	01 010 110	00110100

Fig. 4.22 Sample instructions of 8088

adequate. The third instruction contains 16 bits of immediate data. Note that the low byte of immediate data comes first, followed by its high byte. The fourth assembly statement is identical to the third, however it has been encoded using the immediate to accumulator instruction format. Here, w = 1 implies that the accumulator is the AX register. This instruction is only 3 bytes in length as against the previous instruction which is 4 bytes. This illustrates the fact that the assembler has to perform an analysis of the available options before determining the best instruction to use in a specific case.

Segment overrides

For arithmetic and `MOV` instructions, the architecture uses the data segment by default. To override this, an instruction can be preceded by a 1-byte segment override prefix with the following format:

001 *seg* 110

where *seg*, represented in 2 bits, has the meanings shown in Fig. 4.23.

seg	*segment register*
00	ES
01	CS
10	SS
11	DS

Fig. 4.23 Segment codes

Example 4.8 If the code segment is to be used instead of the data segment in the second statement of Fig. 4.22, it can be rewritten as

```
ADD        AL, CS:12H[SI]
```

The assembler would encode this as

segment override	*instruction*
001 01 110	000000 1 0 01 000 100 00010010

Control transfer instructions

Two groups of control transfer instructions are supported. These are:

1. Calls, jumps and returns
2. Iteration control instructions.

The calls, jumps and returns can occur within the same segment, or can cross segment boundaries. Intra-segment transfers are preferably assembled using a self-relative displacement in the range of −128 to +127. The longer form of intra-segment transfer uses a 16 bit logical address within the segment. Inter-segment transfers indicate a new segment base and an offset. Their execution is expensive since the code segment register has to be modified. Control transfers can be both direct and indirect. Their instruction formats are as shown in Fig. 4.24.

(a) *Intrasegment*

Opcode	*Disp. low*	*Disp. high*

(b) *Intersegment*

Opcode	*Offset*	*Offset*	*Segment*	*base*

(c) *Indirect*

Opcode	*mod 100 r/m*	*Disp. low*	*Disp. high*

Fig. 4.24 Formats of control transfer instructions

A call pushes the offset of the next instruction's address from the segment base on the stack. This address is used to return to the calling program. In the case of an inter-segment call, the CS segment register is pushed first, followed by the offset.

Iteration control operations perform looping decisions in string operations. A self-relative jump of −128 to +127 permits the decision to be made at the beginning or end of the loop.

Example 4.9 Consider the program

```
MOV     SI, 100H   ; Source address
MOV     DI, 200H   ; Destination address
MOV     CX, 50H    ; No.  of bytes
CLD                ; Clear direction flag
REP     MOVSB      ; Move 80 bytes
```

The MOVSB instruction moves one byte from the address contained in SI to the address contained in DI. After the move, the SI and DI registers are incremented/decremented by one byte depending on the direction flag. Since the CLD instruction resets the direction flag, the registers will be incremented in this case. The REP prefix of MOVSB

will cause CX to be decremented after MOVSB is performed. The new value of CX is tested and the loop is repeated until CX becomes zero.

4.5.3 The Assembly Language of Intel 8088

Statement format

The format of an assembly statement is as follows:

```
[Label:]  opcode     operand(s)  ; comment string
```

where the label is optional. In the operand field, operands are separated by commas. Figure 4.22 contains some examples of assembly statements. The parentheses [..] in the operand field represent the words 'contents of'. Base and index register specifications, as also direct addresses specified as numeric offsets from the segment base are enclosed in these parentheses. A segment override is specified in the operand to which it applies, viz. CS:12H [SI]. More details of operand specification can be found in Sections 4.5.1 and 4.5.2.

Assembler directives

The ORG, EQU and END directives are analogous to the ORIGIN, EQU and END directives described in Sections 4.4 and 4.1. The start directive is not supported since ORG subsumes its functionality. The concept of literals, and the LTORG directive, are redundant since the 8088 architecture supports immediate operands.

Declarations

Declaration of constants and reservation of storage are both achieved in the same directive, viz.

```
A    DB    25        ;Reserve byte & initialize
B    DW    ?         ;Reserve word, no initialization
C    DD    6DUP(0)   ;6 Double words, all 0's
```

DQ and DT reserve a quad-word (8 bytes) and ten bytes respectively. Since logical addresses (16 bits) are required to occupy a word, the DW declaration is used for some special purposes, viz.

```
ADDR_A    DW        A
```

initializes the word to the logical address of A (i.e. offset from the segment base).

EQU *and* PURGE

EQU defines symbolic names to represent values or other symbolic names as described in Section 4.4. The names so defined can be 'undefined' through a PURGE statement. Such a name can be reused for other purposes later in the program.

Example 4.10 Following program illustrates EQU and PURGE.

```
XYZ     DB      ?
ABC     EQU     XYZ   ; ABC represents name XYZ
        PURGE   ABC   ; ABC no longer XYZ
ABC     EQU     25    ; ABC now stands for '25'
```

SEGMENT, ENDS *and* ASSUME

All memory addressing is segment directed. Hence as assembly program consists of a set of segments. SEGMENT and ENDS directives demarcate the segments in an assembly program. To assemble a symbolic reference, the assembler must determine the offset of the symbol from the start of the segment containing it. To facilitate this, the programmer must perform the following actions in the assembly program: (a) load a segment register with the segment base, and (b) let the assembler know which segment register contains the segment base. The second task is performed using the ASSUME directive, which has the syntax

ASSUME *<register>* : *<segment name>*

and tells the assembler that it can 'assume' the address of the indicated segment to be present in *<register>*. The directive ASSUME *<register>* : NOTHING cancels any prior assumptions indicated for *<register>*.

Example 4.11 Consider the following program:

```
SAMPLE_DATA   SEGMENT
ARRAY         DW          100 DUP ?
SUM           DW          0
SAMPLE_DATA   ENDS
SAMPLE_CODE   SEGMENT

              ASSUME      DS:SAMPLE_DATA
HERE:         MOV         AX, SAMPLE_DATA
              MOV         DS, AX
              MOV         AX, SUM
              - -
SAMPLE_CODE   ENDS
              END         HERE
```

The program consists of two segments, a code segment and a data segment. The ASSUME directive tells the assembler that the start address of SAMPLE_DATA can be assumed to be in the DS register. While assembling the statement MOV AX, SUM the assembler first computes the offset of SUM from the start of its segment. This is 200 bytes. It now finds whether the segment in which SUM exists is addressable at the current moment. Since it is, it would encode SUM to be an offset of 200 bytes from the DS

register. Note that it is the programmer's responsibility to ensure that the correct address is loaded in the DS register before executing the reference to SUM. If the address of SAMPLE_DATA were to be loaded into some other segment register, e.g. register ES, this fact would be indicated through the statement ASSUME ES:SAMPLE_DATA. The assembler would then generate a segment override prefix while assembling the statement MOV AX, SUM.

PROC, ENDP, NEAR *and* FAR

PROC and ENDP delimit the body of a procedure. The keywords NEAR and FAR appearing in the operand field of PROC indicate whether the call to the procedure is to be assembled as a *near* or *far* call. A RET statement must appear in the body of the procedure to return execution control to the calling program. Parameters for the called procedure can be passed through registers or on the stack.

Example 4.12 Consider the following assembly program:

```
SAMPLE_CODE  SEGMENT
CALCULATE    PROC        FAR          ; a FAR procedure
             - -
             RET
CALCULATE    ENDP
SAMPLE_CODE  ENDS
PGM          SEGMENT
             - -
             CALL        CALCULATE    ; a FAR call
             - -
PGM          ENDS
             END
```

Since CALCULATE is a far procedure, it need not be addressable at the point of call. The assembler will encode a far call instruction which specifies the segment base and the offset of CALCULATE within the segment.

PUBLIC *and* EXTRN

When a symbolic name declared in one assembly module is to be accessible in other modules, it is specified in a PUBLIC statement. Another module wishing to use this name must specify it in an EXTRN statement which has the syntax

EXTRN <*symbolic name*>:<*type*>

For labels of DC, DS statements, the type can be word, byte, etc. For labels of instructions, the type can be FAR or NEAR.

Analytic operators

The analytic operators split a memory address into its components, or provide information regarding the type and memory requirements of operands. Five analytic

operators exist. These are SEG, OFFSET, TYPE, SIZE and LENGTH. SEG and OFFSET provide the segment and offset components of the memory address of an operand.

Example 4.13 The instruction

```
MOV        AX, OFFSET ABC
```

loads the offset of symbol ABC within its segment into the AX register.

TYPE indicates the manner in which an operand is defined and returns the following numeric codes: 1 (byte), 2 (word), 4 (double word), 8 (quad-word), 10 (ten bytes), −1 (near instruction) and −2 (far instruction). SIZE indicates the number of units declared for an operand, while LENGTH indicates the number of bytes allocated to the operand.

Example 4.14 The symbol BUFFER defined in

```
BUFFER     DW         100 DUP (0)
```

has the SIZE of 100 and LENGTH of 200 bytes. SIZE and LENGTH can be used as in

```
MOV        CX, LENGTH XYZ
```

which loads length of XYZ into the CX register.

Synthetic operators

It is sometimes necessary to have different types associated with the same memory operand, e.g. when a byte in an operand of type 'word' is to be accessed as an operand of type 'byte'. This is achieved through the PTR and THIS operators. The PTR operator creates a new memory operand with the same segment and offset addresses as an existing operand, but having a different type. No memory allocation is implied by its use. The THIS operator performs the special function of creating a new memory operand with the same address as the next memory byte available for allocation.

Example 4.15 Consider the program

```
XYZ        DW         312
NEW_NAME   EQU        BYTE PTR XYZ
LOOP:      CMP        AX, 234
           JMP        LOOP
FAR_LOOP   EQU        FAR PTR LOOP
           JMP        FAR_LOOP
```

Here, NEW_NAME is a byte operand with the same address as XYZ, while FAR_LOOP is a FAR symbolic name with the same address as LOOP. Thus, while JMP LOOP is a near jump, JMP FAR_LOOP is a far jump. Exactly the same effect could be achieved by rewriting the program using THIS as follows:

```
                DW
NEW_NAME        EQU         THIS BYTE
XYZ             DW          312
FAR_LOOP        EQU         THIS FAR
LOOP            CMP         AX, 234
                JMP         LOOP
                -- --
                JMP         FAR_LOOP
```

4.5.4 Problems of Single Pass Assembly

A single pass assembler for Intel 8088 shares some problems with other single pass assemblers, viz. problems in assembling forward references and in error reporting. The forward reference problem is aggravated by the nature of the 8088 architecture. We discuss two aspects of this problem. The sample program of Fig. 4.25 is used to illustrate these problems.

```
Sr. No.                          Statement                    Offset
001     CODE    SEGMENT
002             ASSUME    CS:CODE, DS:DATA
003             MOV       AX, DATA                            0000
004             MOV       DS, AX                              0003
005             MOV       CX, LENGTH STRNG                    0005
006             MOV       COUNT,0000                          0008
007             MOV       SI, OFFSET STRNG                    0011
008             ASSUME    ES:DATA, DS:NOTHING
009             MOV       AX, DATA                            0014
010             MOV       ES, AX                              0017
011     COMP:   CMP       [SI],'A'                            0019
012             JNE       NEXT                                0022
013             MOV       COUNT, 1                            0024
014     NEXT:   INC       SI                                  0027
015             DEC       CX                                  0029
016             JNE       COMP                                0030
017     CODE    ENDS

018     DATA    SEGMENT
019             ORG       1
020     COUNT   DB        ?                                   0001
021     STRNG   DW        50 DUP (?)                          0002
022     DATA    ENDS
023             END
```

Fig. 4.25 Sample assembly program of Intel 8088

Forward references

A symbolic name may be forward referenced in a variety of ways. When used as a data operand in a statement, its assembly is straightforward. An entry can be made in the table of incomplete instructions (TII) discussed in Section 4.3. This entry would identify the bytes in code where the address of the referenced symbol should be put. When the symbol's definition is encountered, this entry would be analysed to complete the instruction. However, use of a symbolic name as the destination in a branch instruction gives rise to a peculiar problem. Some generic branch opcodes like JMP in the 8088 assembly language can give rise to instructions of different formats and different lengths depending on whether the jump is *near* or *far*—that is, whether the destination symbol is less than 128 bytes away from the JMP instruction. However, this would not be known until sometime later in the assembly process! This problem is solved by assembling such instructions with a 16 bit logical address unless the programmer indicates a short displacement, e.g. JMP SHORT LOOP. The program of Fig. 4.25 contains the forward branch instruction JNE NEXT. However, the above problem does not arise here since the opcode JNE dictates that the instruction should be in the self-relative format.

A more serious problem arises when the type of a forward referenced symbol is used in an instruction. The type may be used in a manner which influences the size/length of a declaration. Such usage will have to be disallowed to facilitate single pass assembly.

Example 4.16 Consider the statements

```
XYZ     DB      LENGTH ABC DUP(0)
- -
ABC     DD      ?
```

Here the forward reference to ABC makes it impossible to assemble the DB statement in a single pass.

Segment registers

An ASSUME statement indicates that a segment register contains the base address of a segment. The assembler represents this information by a pair of the form (*segment register, segment name*). This information can be stored in a *segment registers table* (SRTAB). SRTAB is updated on processing an ASSUME statement. For processing the reference to a symbol *symb* in an assembly statement, the assembler accesses the symbol table entry of *symb* and finds (seg_{symb}, $offset_{symb}$) where seg_{symb} is the name of the symbol containing the definition of *symb*. It uses the information in SRTAB to find the register which contains seg_{symb}. Let it be register r. It now synthesizes the pair (r, $offset_{symb}$). This pair is put in the address field of the target instruction.

However, this strategy would not work while assembling forward references. Consider statements 6 and 13 in Fig. 4.25 which make forward references to COUNT.

When the definition of COUNT is encountered in statement 20, information concerning these forward references can be found in the table of incomplete instructions (TII). What segment register should be used to assemble these references? The first reference was made in statement 6 when DS was the segment register containing the segment base of DATA. However, SRTAB presently contains the pair (ES, DATA) as a result of statement 8, viz. ASSUME ES:DATA . . . A similar problem may arise while assembling forward references contained in branch instructions. The following provisions are made to handle this problem:

1. A new SRTAB is created while processing an ASSUME statement. This SRTAB differs from the old SRTAB only in the entries for the segment registers named in the ASSUME statement. Since many SRTAB's exist at any time, an array named SRTAB_ARRAY is used to store the SRTAB's. This array is indexed using a counter *srtab_no*.
2. Instead of TII, a *forward reference table* (FRT) is used. Each entry of FRT contains the following entries:
   (a) Address of the instruction whose operand field contains the forward reference
   (b) Symbol to which forward reference is made
   (c) Kind of reference (e.g. T : analytic operator TYPE, D : data address, S : self relative address, L : length, F : offset, etc.)
   (d) Number of the SRTAB to be used for assembling the reference.

Example 4.17 illustrates how these provisions are adequate to handle the problem concerning forward references mentioned earlier.

Example 4.17 Two SRTAB's would be built for the program of Fig. 4.25. SRTAB#1 contains the pairs (CS, CODE) and (DS, DATA) while SRTAB#2 contains the pairs (CS, CODE) and (ES, DATA). While processing statement 6, SRTAB#1 is the current SRTAB. Hence the FRT entry for this statement is (008, COUNT, D, SRTAB#1). Similarly the FRT entry for statement 13 is (024, COUNT, D, SRTAB#2). These entries are processed on encountering the definition of COUNT, giving the address pairs (DS, 001) and (ES, 001). (Note that FRT entries would also exist for statements 5, 7 and 12. However, none of them require the use of a base register.)

4.5.5 Design of the Assembler

Algorithm for the Intel 8088 assembler, Algorithm 4.3, is given at the end of this section. LC processing in this algorithm differs from LC processing in the first pass of a two pass assembler (see Algorithm 4.1) in one significant respect. In Intel 8088, the unit for memory allocation is a byte, however certain entities require their starting byte to be aligned on specific boundaries in the address space. For example, a word requires alignment on an even boundary, i.e. it must have an even start address. Such alignment requirements may force some bytes to be left unused during memory allocation. Hence while processing DB statements and imperatives, assembler first aligns

LC on the requisite boundary. We call this *LC alignment*. Allocation of memory for a statement and entering its label in the symbol table is performed after LC alignment.

The data structures of the assembler are illustrated in Fig. 4.26, where numbers in parentheses indicate the number of bytes required for a field. The mnemonics table (MOT) is hash organized and contains the following fields: *mnemonic opcode, machine opcode, alignment/format info* and *routine id*. The *routine id* field of an entry specifies the routine which processes that opcode. *Alignment/format info* is specific to a given routine. For example, the code of '00H' for routine R2 implies that only one instruction format, that with self-relative displacement, is supported. 'FFH' for the same routine implies that all formats are supported, hence the routine must decide which machine opcode to use. The symbol table (SYMTAB) is also hash-organized and contains all relevant information about symbols defined and used in the source program. The contents of some important fields are as follows: The *owner segment* field indicates id of the segment in which a symbol is defined. It contains the SYMTAB entry # of the segment name. For a non-EQU symbol the *type* field indicates the alignment information. For an EQU symbol, *type* field indicates whether the symbol is to be given a numeric value or a textual value. The owner segment and offset fields are used to accommodate the value.

An SRTAB can contain upto four entries, one for each register. The current SRTAB exists in the last entry of SRTAB_ARRAY. SRTAB's are accessed by their entry numbers in SRTAB_ARRAY.

Forward references

Information concerning forward references to a symbol is organized in the form of a linked list. Thus, the *forward reference table* (FRT) contains a set of linked lists. The *FRT pointer* field of a SYMTAB entry points to the head of this list. Since ordering of FRT entries is not important, for efficiency reasons new entries are added at the beginning of the list. Each FRT entry contains SRTAB # to be used to assemble the forward reference. It also contains the instruction address and a *usage code* indicating where and how the reference is to be assembled. When the definition of a symbol is encountered, its forward references (if any) are processed, and the forward references list is discarded. This minimizes the size of FRT at any time.

Cross references

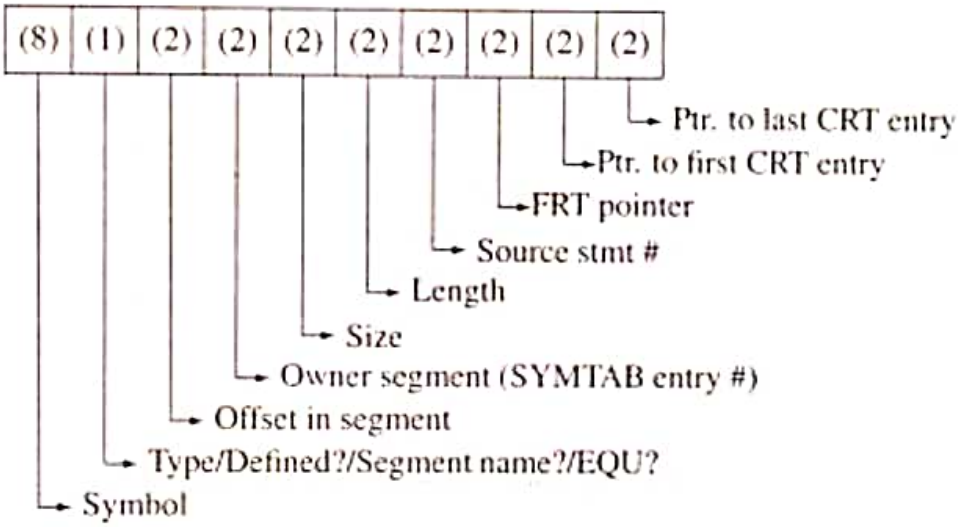
A cross reference directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order of statement numbers. The assembler uses the *cross reference table* (CRT) to collect the information concerning references to all symbols in the program. Each SYMTAB entry points to the head and tail of a linked list in the CRT. New entries are added at the end of the list.

Being linked lists, FRT and CRT can be organized in a single memory area. The tables grow from the high end of storage to its low end. The freed entries of FRT are reused by maintaining a free list. The target code generated by the assembler grows

(a) Mnemonics table (MOT)

Mnemonic opcode (6)	*Machine opcode* (2)	*Alignment/ format info* (1)	*Routine id* (4)
JNE	75H	OOH	R2

(b) Symbol table (Symtab)



(c) Segment Register Table Array (SRTAB_ARRAY)

Segment Register (1)	*Segment name* (2)
00(ES)	23
:	

SRTAB #1

SRTAB #2

(d)Forward Reference table (FRT)

Pointer (2)	*SRTAB #* (1)	*Instruction address* (2)	*Usage code* (1)	*Source stmt #* (2)

(e) Cross Reference table (CRT)

Pointer (2)	*Source Stmt #* (2)

Fig. 4.26 Data structures of the assembler

from the low end to the high end of storage. As a result, no size restrictions need to be placed on individual tables. The assembler fails to handle a source program only if its target code overlaps with its tables.

Example 4.18 Figure 4.27 illustrates contents of important data structures after processing Statement 19 of the source program of Fig. 4.25. The symbol table contains entries for symbols COMP and NEXT whose definitions have already been processed. The *defined* flag of these entries is = 'Yes' and the address and type fields contain appropriate values. NEXT was forward referenced in Statement 12 of the program. An FRT entry was created for this reference with the *usage code* ='S' to indicate that a self-relative displacement is desired. When the definition of NEXT was processed (Statement 14), the validity of the forward reference in terms of this requirement was checked and the corresponding instruction was completed. The FRT entry was then discarded.

FRT entries currently exist for symbols COUNT and STRNG. Both references to COUNT in the source program are forward references (Statements 6 and 13). Hence, two entries exist for COUNT in FRT and CRT. The first FRT entry has #1 in the SRTAB field, while the second entry has #2 in it. Similarly two FRT and CRT entries exist for STRNG. The *usage code* fields of the FRT entries indicate what information is required in the referencing instruction, e.g. data address ('D'), self relative address ('S'), length ('L'), offset ('F'), etc. Note that some CRT entries are not shown in Fig. 4.27.

Subsequent processing of the program is as follows: At the end of processing statement 20 (but before incrementing LC), its label, viz. COUNT, will be looked up in SYMTAB. An entry exists for it with *defined* = '*no*'. This implies that COUNT has been forward referenced. Its *segment* and *offset* fields are now set. The forward reference chain is then traversed and each forward reference is processed to perform error detection, and to complete the machine instruction containing the forward reference. The second forward reference to COUNT passes the error detection step and leads to completion of the machine instruction with offset 0024. The first forward reference, however, expects a word alignment for COUNT (since immediate data is 2 bytes in length) which is not the case. An error is indicated at this point. The *FRT pointer* field of COUNT's SYMTAB entry is now reset and the FRT entries for COUNT are destroyed.

Listing and error indication

The program listing and error reporting function faces the problems discussed in Section 4.4.7. The program listing can contain the statement in source form, its serial number in the program and the memory address assigned to it. The target code can only be printed at the end of assembly. Error reporting also suffers due to the single pass assembly scheme. An error cannot be reported against the statement containing a forward reference since the statement would have been already listed out. If the error is simply reported at the end of the source program, it is rather cumbersome for the programmer to identify the erroneous statement.

The following strategy is used to overcome this problem (see Ex. 4.19):

1. The serial number of the source statement containing a forward reference is stored in the FRT entry along with other relevant information (see Fig. 4.27).

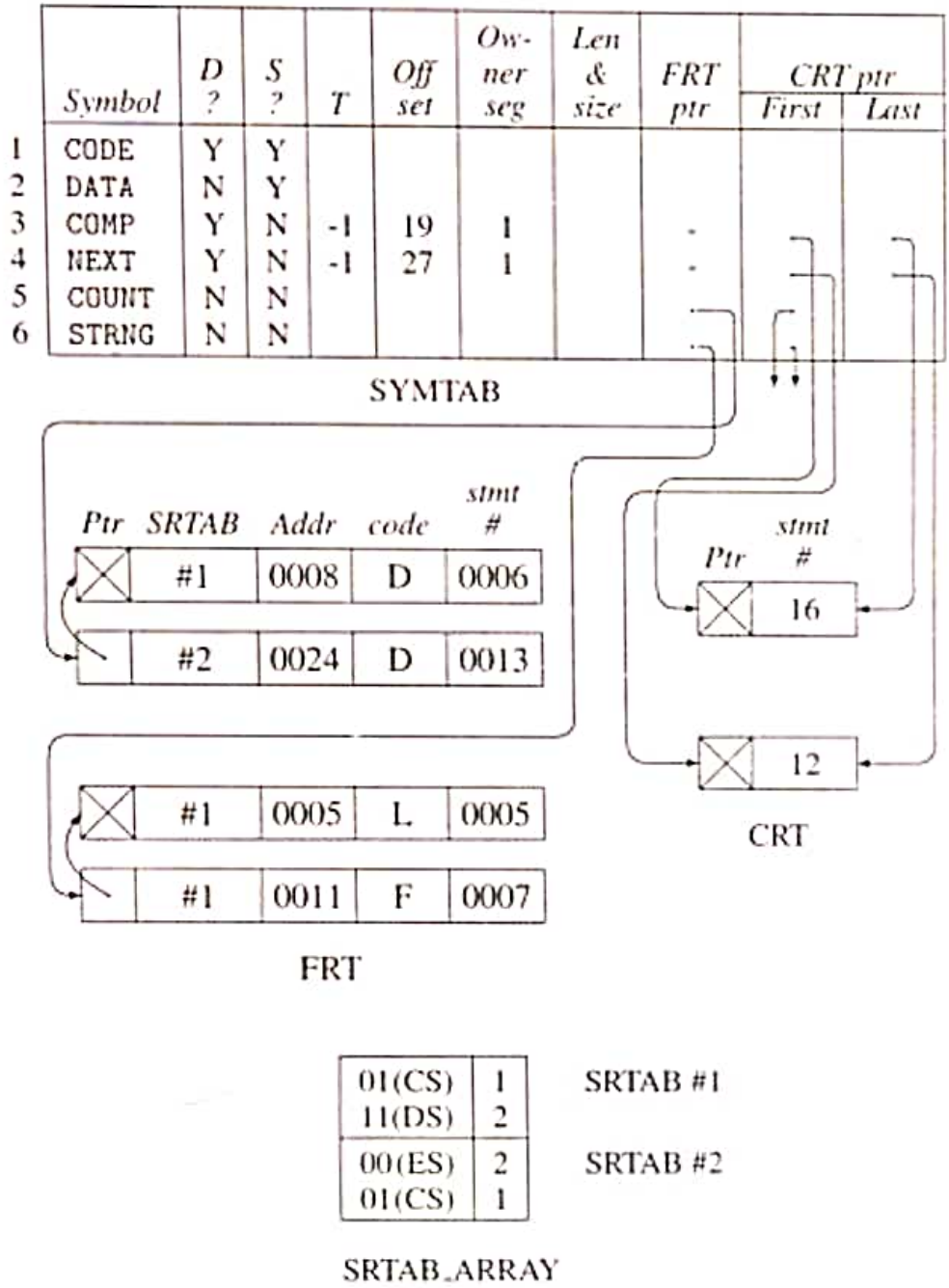


Fig. 4.27 Data structures after processing Statement 19

2. Whenever a symbol's definition is encountered, all forward references to the symbol are processed and errors, if any, are reported against this statement. Though this is not as effective as reporting the error against the erroneous statement, it is the next best thing.

Algorithm 4.3 (Single pass assembler of 8088)

1. *code_area_address* := address of *code_area*;
   srtab_no := 1;
   LC := 0;
   stmt_no := 1;
   SYMTAB_segment_entry := 0;
   Clear ERRTAB, SRTAB_ARRAY.
2. While next statement is not an END statement
   (a) Clear *machine_code_buffer*.
   (b) If label is present then
   this_label := symbol in label field;
   (c) If an EQU statement
   (i) *this_address* := value of operand expression;
   (ii) Make an entry for *this_label* in SYMTAB with
   offset := *this_addr*;
   Defined := 'yes';
   owner_segment := *owner_segment* of operand symbol;
   source_stmt_# := *stmt_no*;
   (iii) Enter *stmt_no* in the CRT list of the label in the operand field.
   (iv) Process forward references to *this_label*;
   (v) *size* := 0;
   (d) If an ASSUME statement
   (i) Copy the SRTAB in SRTAB_ARRAY [*srtab_no*] into SRTAB-_ARRAY [*srtab_no*+1];
   (ii) *srtab_no* := *srtab_no*+1;
   (iii) *this_register* := register mentioned in the statement.
   (iv) *this_segment* := entry number of SYMTAB entry of the segment appearing in operand field.
   (v) Make the entry (*this_register*, *this_segment*) in SRTAB_ARRAY [*srtab_no*]. (This overwrites an existing entry for *this_register*.)
   (vi) *size* := 0;
   (e) If a SEGMENT statement

(i) Make an entry for *this_label* in SYMTAB.

(ii) Set *segment name ?* := *true*;

(iii) *SYMTAB_segment_entry* := entry no. in SYMTAB;

(iv) LC := 0;

(v) *size* := 0;

(f) If an ENDS statement then
SYMTAB_segment_entry := 0;

(g) If a declaration statement

(i) Align LC according to the specification in the operand field.

(ii) Assemble the constant(s), if any, in the *machine_code_buffer*.

(iii) *size* := size of memory area required;

(h) If an imperative statement

(i) If operand is a symbol *symb* then
enter *stmt_no* in CRT list of *symb*.

(ii) If operand symbol is already defined then
Check its alignment and addressibility.
Generate the address specification (segment register, *offset*) for the symbol using its SYMTAB entry and SRTAB_ARRAY [*srtab_no*].
else
Make an entry for *symbol* in SYMTAB.
Defined := '*no*';
Enter (*srtab_no*, LC, *usage code*, *stmt_no*) in FRT.

(iii) Assemble instruction in *machine_code_buffer*.

(iv) *size* := size of the instruction;

(i) If *size* $\neq$ 0 then

(i) If label is present then
Make an entry for *this_label* in SYMTAB.
owner_segment := *SYMTAB_segment_entry*;
Defined := '*yes*';
offset := LC;
source_stmt_# := *stmt_no*;

(ii) Move contents of *machine_code_buffer* to the address *code_area-_address*;

(iii) *code_area_address* := *code_area_address* + *size*;

(iv) Process forward references to the symbol. Check for alignment and addressability errors. Enter errors in ERRTAB.

(v) List the statement with errors contained in ERRTAB.

(vi) Clear ERRTAB.

3. (Processing of END statement)
   (a) Report undefined symbols from SYMTAB.
   (b) Produce cross reference listing.
   (c) Write *code_area* into output file.

Example 4.19 Error in the forward reference to COUNT in Statement 6 of Fig. 4.25 would be reported as:

Stmt no.		*Source statement*		*Offset*	*Instrn*
006		MOV	COUNT, 0000	0005	...
...		...		...	...
020	COUNT	DB	?	0001	...

** error ** Illegal forward reference (alignment) from Stmt 6.

BIBLIOGRAPHY

Most books on computer architecture discuss assembly languages of particular computer systems, e.g. Stone and Siewiorek (1975), Gear (1980), and Macewen (1980). Leeds and Weinberg (1966) is one of the first books devoted to assembly language programming. Flores (1971) covers assembly language of IBM/360. Books by Leventhal deal with assembly language programming for microcomputers.

(a) Assembly language programming

1. Leeds, H.D. and M.W. Weinberg (1966): *Computer Programming Fundamentals*, McGraw-Hill, New York.
2. Leventhal, L.A., A. Osborne, and C. Collins (1980): *Z8000 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley.
3. Rudd, W.G. (1976): *Assembly Language Programming and the IBM 360/370 Computers*, Prentice-Hall, Englewood Cliffs.
4. Stone, H.S. and D.P. Siewiorek (1975): *Introduction to Computer Organization and Data Structures*, McGraw-Hill, New York.
5. Weller, W.J. (1975): *Assembly Level Programming for Small Computers*, Heath & Co., Lexington.
6. Yarmish, R. and J. Yarmish (1979): *Assembly Language Fundamentals 360/370, OS/VS, DOS/VS*, Addison-Wesley, Reading.

(b) Assemblers

1. Barron, D.W. (1969): *Assemblers and Loaders*, Macdonald Elsevier, London.
2. Calingaert, P. (1979): *Assemblers, Compilers and Program Translation*, Computer Science Press, Maryland.
3. Donovan, J.J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
4. Flores, I. (1971): *Assemblers and BAL*, Prentice-Hall, Englewood Cliffs.

CHAPTER 5

Macros and Macro Processors

Macros are used to provide a *program generation* facility (see Section 1.2.1) through *macro expansion*. Many languages provide built-in facilities for writing macros. Well known examples of these are the higher level languages PL/I, C, Ada and C++. Assembly languages of most computer systems also provide such facilities. When a language does not support built-in macro facilities, a programmer may achieve an equivalent effect by using generalized preprocessors or software tools like Awk of Unix. The discussion in this chapter is confined to macro facilities provided in assembly languages.

Definition 5.1 (Macro) *A* macro *is a unit of specification for program generation through expansion.*

A macro consists of a name, a set of formal parameters and a body of code. The use of a macro name with a set of actual parameters is replaced by some code generated from its body. This is called *macro expansion*. Two kinds of expansion can be readily identified:

1. *Lexical expansion:* Lexical expansion implies replacement of a character string by another character string during program generation. Lexical expansion is typically employed to replace occurrences of formal parameters by corresponding actual parameters.
2. *Semantic expansion:* Semantic expansion implies generation of instructions tailored to the requirements of a specific usage—for example, generation of type specific instructions for manipulation of byte and word operands. Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in the number, sequence and opcodes of instructions.

Example 5.1 The following sequence of instructions is used to increment the value in a memory word by a constant:

1. Move the value from the memory word into a machine register.
2. Increment the value in the machine register.
3. Move the new value into the memory word.

Since the instruction sequence MOVE-ADD-MOVE may be used a number of times in a program, it is convenient to define a macro named INCR. Using lexical expansion the macro call INCR A, B, AREG can lead to the generation of a MOVE-ADD-MOVE instruction sequence to increment A by the value of B using AREG to perform the arithmetic.

Use of semantic expansion can enable the instruction sequence to be adapted to the types of A and B. For example, for Intel 8088, an INC instruction could be generated if A is a byte operand and B has the value '1', while a MOV-ADD-MOV sequence can be generated in all other situations.

Note that macros differ from subroutines in one fundamental respect. Use of a macro name in the mnemonic field of an assembly statement leads to its *expansion*, whereas use of a subroutine name in a call instruction leads to its *execution*. Thus, programs using macros and subroutines differ significantly in terms of program size and execution efficiency. In fact, macros can be said to trade program size for execution efficiency of a program. Other differences between macros and subroutines will be discussed along with the discussion of advanced macro facilities in Section 5.4.

5.1 MACRO DEFINITION AND CALL

Macro definition

A *macro definition* is enclosed between a *macro header* statement and a *macro end* statement. Macro definitions are typically located at the start of a program. A macro definition consists of

1. A *macro prototype* statement
2. One or more *model statements*
3. Macro *preprocessor statements*.

The macro prototype statement declares the name of a macro and the names and kinds of its parameters. A model statement is a statement from which an assembly language statement may be generated during macro expansion. A preprocessor statement is used to perform auxiliary functions during macro expansion.

The macro prototype statement has the following syntax:

$$<macro\ name> \quad [<formal\ parameter\ spec>\ [,..]]$$

where <*macro name*> appears in the mnemonic field of an assembly statement and <*formal parameter spec*> is of the form

$$\&<parameter\ name>[<parameter\ kind>] \tag{5.1}$$

Macro call

A macro is called by writing the macro name in the mnemonic field of an assembly statement. The macro call has the syntax

$$<macro\ name> \quad [<actual\ parameter\ spec>\ [,..]] \tag{5.2}$$

where an actual parameter typically resembles an operand specification in an assembly language statement.

Example 5.2 Figure 5.1 shows the definition of macro INCR. MACRO and MEND are the macro header and macro end statements, respectively. The prototype statement indicates that three parameters called MEM_VAL, INCR_VAL and REG exist for the macro. Since parameter kind is not specified for any of the parameters, they are all of the default kind 'positional parameter'. Statements with the operation codes MOVER, ADD and MOVEM are model statements. No preprocessor statements are used in this macro.

```
MACRO
INCR        &MEM_VAL, &INCR_VAL, &REG
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig. 5.1 A macro definition

5.2 MACRO EXPANSION

A macro call leads to *macro expansion*. During macro expansion, the macro call statement is replaced by a sequence of assembly statements. To differentiate between the original statements of a program and the statements resulting from macro expansion, each expanded statement is marked with a '+' preceding its label field.

Two key notions concerning macro expansion are:

1. *Expansion time control flow:* This determines the order in which model statements are visited during macro expansion.
2. *Lexical substitution:* Lexical substitution is used to generate an assembly statement from a model statement.

Flow of control during expansion

The default flow of control during macro expansion is *sequential*. Thus, in the absence of preprocessor statements, the model statements of a macro are visited sequentially starting with the statement following the macro prototype statement and ending with the statement preceding the MEND statement. A preprocessor statement can alter the flow of control during expansion such that some model statements

are either never visited during expansion, or are repeatedly visited during expansion. The former results in *conditional expansion* and the latter in *expansion time loops*.

The flow of control during macro expansion is implemented using a *macro expansion counter* (MEC).

Algorithm 5.1 (Outline of macro expansion)

1. MEC := statement number of first statement following the prototype statement;
2. While statement pointed by MEC is not a MEND statement
   (a) If a model statement then
      (i) Expand the statement.
      (ii) MEC := MEC + 1;
   (b) Else (i.e. a preprocessor statement)
      (i) MEC := new value specified in the statement;
3. Exit from macro expansion.

MEC is set to point at the statement following the prototype statement. It is incremented by 1 after expanding a model statement. Execution of a preprocessor statement can set MEC to a new value to implement conditional expansion or expansion time loops. These features are discussed in Section 5.4.

Lexical substitution

A model statement consists of 3 types of strings

1. An ordinary string, which stands for itself.
2. The name of a formal parameter which is preceded by the character '&'.
3. The name of a preprocessor variable, which is also preceded by the character '&'.

During lexical expansion, strings of type 1 are retained without substitution. Strings of types 2 and 3 are replaced by the 'values' of the formal parameters or preprocessor variables. The value of a formal parameter is the corresponding actual parameter string. The rules for determining the value of a formal parameter depend on the kind of parameter.

Positional parameters

A positional formal parameter is written as &*<parameter name>*, e.g. &SAMPLE where SAMPLE is the name of a parameter. In other words, *<parameter kind>* of syntax rule (5.1) is omitted. The *<actual parameter spec>* in a call on a macro using positional parameters [see syntax rule (5.2)] is simply an *<ordinary string>*.

The value of a positional formal parameter XYZ is determined by the rule of *positional association* as follows:

1. Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.
2. Find the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement (see syntax rule (5.2)). Let this be the ordinary string ABC. Then, the value of formal parameter XYZ is ABC.

Example 5.3 Consider the call

```
INCR        A, B, AREG
```

on macro INCR of Fig. 5.1. Following the rule of positional association, values of the formal parameters are:

formal parameter	*value*
MEM_VAL	A
INCR_VAL	B
REG	AREG

Lexical expansion of the model statements now leads to the code

```
+        MOVER       AREG, A
+        ADD         AREG, B
+        MOVEM       AREG, A
```

Keyword parameters

For keyword parameters, <*parameter name*> is an ordinary string and <*parameter kind*> is the string '=' in syntax rule (5.1). The <*actual parameter spec*> is written as <*formal parameter name*> = <*ordinary string*>. The value of a formal parameter XYZ is determined by the rule of *keyword association* as follows:

1. Find the actual parameter specification which has the form XYZ= <*ordinary string*>.
2. Let <*ordinary string*> in the specification be the string ABC. Then the value of formal parameter XYZ is ABC.

Note that the ordinal position of the specification XYZ=ABC in the list of actual parameters is immaterial. This is very useful in situations where long lists of parameters have to be used.

Example 5.4 Figure 5.2 shows macro INCR of Fig. 5.1 rewritten as macro INCR_M using keyword parameters. The following macro calls

```
INCR_M      MEM_VAL=A, INCR_VAL=B, REG=AREG
...
INCR_M      INCR_VAL=B, REG=AREG, MEM_VAL=A
```

are now equivalent.

```
MACRO
INCR_M      &MEM_VAL=, &INCR_VAL=, &REG=
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig. 5.2 A macro definition using keyword parameters

Default specifications of parameters

A *default* is a standard assumption in the absence of an explicit specification by the programmer. Default specification of parameters is useful in situations where a parameter has the same value in most calls. When the desired value is different from the default value, the desired value can be specified explicitly in a macro call. This specification overrides the default value of the parameter for the duration of the call.

Default specification of keyword parameters can be incorporated by extending syntax (5.1), the syntax for formal parameter specification, as follows:

$$\&<\textit{parameter name}>[<\textit{parameter kind}>[<\textit{default value}>]] \qquad (5.3)$$

Example 5.5 Register AREG is used for all arithmetic in a program. Hence most calls on macro INCR_M contain the specification &REG=AREG. The macro can be redefined to use a default specification for the parameter REG shown in Fig. 5.3 (see macro INCR_D). Consider the following calls

```
INCR_D      MEM_VAL=A, INCR_VAL=B
INCR_D      INCR_VAL=B, MEM_VAL=A
INCR_D      INCR_VAL=B, MEM_VAL=A, REG=BREG
```

The first two calls are equivalent to the calls in Ex. 5.4. The third call overrides the default value for REG with the value BREG. BREG will be used to perform the arithmetic in its expanded code.

```
MACRO
INCR_D      &MEM_VAL=, &INCR_VAL=, &REG=AREG
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig. 5.3 A macro definition with default parameter

Macros with mixed parameter lists

A macro may be defined to use both positional and keyword parameters. In such a case, all positional parameters must precede all keyword parameters. For example, in the macro call

```
SUMUP      A,B,G=20,H=X
```

A, B are positional parameters while G, H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately.

Other uses of parameters

The model statements of Examples 5.2-5.5 have used formal parameters only in operand fields. However, use of parameters is not restricted to these fields. Formal parameters can also appear in the label and opcode fields of model statements.

Example 5.6

```
        MACRO
        CALC        &X, &Y, &OP= MULT, &LAB=

&LAB    MOVER       AREG, &X
        &OP         AREG, &Y
        MOVEM       AREG, &X
        MEND
```

Expansion of the call `CALC A, B, LAB=LOOP` leads to the following code:

```
+ LOOP   MOVER      AREG, A
+        MULT       AREG, B
+        MOVEM      AREG, A
```

5.3 NESTED MACRO CALLS

A model statement in a macro may constitute a call on another macro. Such calls are known as *nested macro calls*. We refer to the macro containing the nested call as the *outer* macro and the called macro as the *inner* macro. Expansion of nested macro calls follows the *last-in-first-out* (LIFO) rule. Thus, in a structure of nested macro calls, expansion of the latest macro call (i.e. the innermost macro call in the structure) is completed first.

Example 5.7 Macro COMPUTE of Fig. 5.4 contains a nested call on macro INCR_D of Fig. 5.3. Figure 5.5 shows the expanded code for the call

```
COMPUTE     X, Y
```

After lexical expansion, the second model statement of COMPUTE is recognized to be a call on macro INCR_D. Expansion of this macro is now performed. This leads to generation of statements marked [2], [3] and [4] in Fig. 5.5. The third model statement of COMPUTE is now expanded. Thus the expanded code for the call on COMPUTE is:

```
+          MOVEM       BREG, TMP
+          MOVER       BREG, X
+          ADD         BREG, Y
+          MOVEM       BREG, X
+          MOVER       BREG, TMP

MACRO
COMPUTE      &FIRST, &SECOND
MOVEM        BREG, TMP
INCR_D       &FIRST, &SECOND, REG=BREG
MOVER        BREG, TMP
MEND
```

Fig. 5.4 A nested macro call

```
                                                  + MOVER   BREG,X [2]
                 + MOVEM   BREG,TMP [1]           + ADD     BREG,Y [3]
COMPUTE  X,Y     + INCR_D  X,Y                    + MOVEM   BREG,X [4]
                 + MOVER   BREG,TMP [5]
```

Fig. 5.5 Expanded code for a nested macro call

5.4 ADVANCED MACRO FACILITIES

Advanced macro facilities are aimed at supporting semantic expansion. These facilities can be grouped into

1. Facilities for alteration of flow of control during expansion
2. Expansion time variables
3. Attributes of parameters.

This section describes some advanced facilities and illustrates their use in performing conditional expansion of model statements and in writing expansion time loops.

Alteration of flow of control during expansion

Two features are provided to facilitate alteration of flow of control during expansion:

1. Expansion time sequencing symbols
2. Expansion time statements AIF, AGO and ANOP.

A sequencing symbol (SS) has the syntax

$$.\ <\textit{ordinary string}> \tag{5.4}$$

As SS is defined by putting it in the label field of a statement in the macro body. It is used as an operand in an AIF or AGO statement to designate the destination of an expansion time control transfer. It never appears in the expanded form of a model statement.

An AIF statement has the syntax

AIF (<*expression*>) <*sequencing symbol*>

where <*expression*> is a relational expression involving ordinary strings, formal parameters and their attributes, and expansion time variables. If the relational expression evaluates to *true*, expansion time control is transferred to the statement containing <*sequencing symbol*> in its label field. An AGO statement has the syntax

AGO <*sequencing symbol*>

and unconditionally transfers expansion time control to the statement containing <*sequencing symbol*> in its label field. An ANOP statement is written as

<*sequencing symbol*> ANOP

and simply has the effect of defining the sequencing symbol.

Expansion time variables

Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls. A local EV is created for use only during a particular macro call. A global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it. Local and global EV's are created through declaration statements with the following syntax:

LCL <*EV specification*>[,<*EV specification*> ..]
GBL <*EV specification*>[,<*EV specification*> ..]

and <*EV specification*> has the syntax &<*EV name*>, where <*EV name*> is an ordinary string.

Values of EV's can be manipulated through the preprocessor statement SET. A SET statement is written as

<*EV specification*> SET <*SET-expression*>

where <*EV specification*> appears in the label field and SET in the mnemonic field. A SET statement assigns the value of <*SET-expression*> to the EV specified in <*EV specification*>. The value of an EV can be used in any field of a model statement, and in the expression of an AIF statement.

Example 5.8

```
        MACRO
        CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

A call on macro CONSTANTS is expanded as follows: The local EV A is created. The first SET statement assigns the value '1' to it. The first DB statement thus declares a byte constant '1'. The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

Attributes of formal parameters

An attribute is written using the syntax

<attribute name>'<formal parameter spec>

and represents information about the value of the formal parameter, i.e. about the corresponding actual parameter. The type, length and size attributes have the names T, L and S.

Example 5.9

```
        MACRO
        DCL_CONST   &A
        AIF         (L'&A EQ 1) .NEXT
        - -
.NEXT   - -
        - -
        MEND
```

Here expansion time control is transferred to the statement having .NEXT in its label field only if the actual parameter corresponding to the formal parameter A has the length of '1'.

5.4.1 Conditional Expansion

While writing a general purpose macro it is important to ensure execution efficiency of its generated code. Conditional expansion helps in generating assembly code specifically suited to the parameters in a macro call. This is achieved by ensuring that a model statement is visited only under specific conditions during the expansion of a macro. The AIF and AGO statements are used for this purpose.

Example 5.10 It is required to develop a macro EVAL such that a call

```
EVAL        A, B, C
```

generates efficient code to evaluate A−B+C in AREG. When the first two parameters of a call are identical, EVAL should generate a single MOVER instruction to load the 3rd parameter into AREG. This is achieved as follows:

```
        MACRO
        EVAL    &X, &Y, &Z
        AIF     (&Y EQ &X) .ONLY
        MOVER   AREG, &X
        SUB     AREG, &Y
        ADD     AREG, &Z
        AGO     .OVER
.ONLY   MOVER   AREG, &Z
.OVER   MEND
```

Since the value of a formal parameter is simply the corresponding actual parameter, the AIF statement effectively compares names of the first two actual parameters. If the names are same, expansion time control is transferred to the model statement `MOVER AREG, &Z`. If not, the MOVE-SUB-ADD sequence is generated and expansion time control is transferred to the statement `.OVER MEND` which terminates the expansion. Thus efficient code is generated under all conditions.

Expansion time loops

It is often necessary to generate many similar statements during the expansion of a macro. This can be achieved by writing similar model statements in the macro.

Example 5.11

```
MACRO
CLEAR   &A
MOVER   AREG, ='0'
MOVEM   AREG, &A
MOVEM   AREG, &A+1
MOVEM   AREG, &A+2
MEND
```

When called as `CLEAR B`, the `MOVER` statement puts the value '0' in AREG, while the three `MOVEM` statements store this value in 3 consecutive bytes with the addresses B, B+1 and B+2.

Alternatively, the same effect can be achieved by writing an expansion time loop which visits a model statement, or a set of model statements, repeatedly during macro expansion. Expansion time loops can be written using expansion time variables (EV's) and expansion time control transfer statements AIF and AGO.

Example 5.12

```
            MACRO
            CLEAR       &X, &N
            LCL         &M
&M          SET         0
            MOVER       AREG, ='0'
.MORE       MOVEM       AREG, &X+&M
&M          SET         &M+1
            AIF         (&M NE N) .MORE
            MEND
```

Consider expansion of the macro call

```
CLEAR       B, 3
```

The LCL statement declares M to be a local EV. At the start of expansion of the call, M is initialized to zero. The expansion of model statement MOVEM AREG, &X+&M thus leads to generation of the statement MOVEM AREG, B. The value of M is incremented by 1 and the model statement MOVEM .. is expanded repeatedly until its value equals the value of N, which is 3 in this case. Thus the macro call leads to generation of the statements

```
+           MOVER       AREG, ='0'
+           MOVEM       AREG, B
+           MOVEM       AREG, B+1
+           MOVEM       AREG, B+2
```

Comparison with execution time loops

Most expansion time loops can be replaced by execution time loops. For example, instead of generating many MOVEM statements as in Ex. 5.12 to clear the memory area starting on B, it is possible to write an execution time loop which moves 0 into B, B+1 and B+2. An execution time loop leads to more compact assembly programs. However, such programs would execute slower than programs containing expansion time loops. Thus a macro can be used to trade program size for execution efficiency.

5.4.2 Other Facilities for Expansion Time Loops

Many assemblers provide other facilities for conditional expansion, an ELSE clause in AIF being an obvious example. The assemblers for M 68000 and Intel 8088 processors provide explicit expansion time looping constructs. We discuss two such facilities here.

The REPT statement

REPT <*expression*>

<expression> should evaluate to a numerical value during macro expansion. The statements between REPT and an ENDM statement would be processed for expansion *<expression>* number of times. Example 5.13 illustrates the use of this facility to declare 10 constants with the values 1, 2, .. 10.

Example 5.13

```
        MACRO
        CONST10
        LCL     &M
&M      SET     1
        REPT    10
        DC      '&M'
&M      SETA    &M+1
        ENDM
        MEND
```

The IRP statement

IRP *<formal parameter>*, *<argument-list>*

The formal parameter mentioned in the statement takes successive values from the argument list. For each value, the statements between the IRP and ENDM statements are expanded once.

Example 5.14

```
MACRO
CONSTS      &M, &N, &Z
IRP         &Z, &M, 7, &N
DC          '&Z'
ENDM
MEND
```

A macro call `CONSTS  4, 10` leads to declaration of 3 constants with the values 4, 7 and 10.

5.4.3 Semantic Expansion

Semantic expansion is the generation of instructions tailored to the requirements of a specific usage. It can be achieved by a combination of advanced macro facilities like AIF, AGO statements and expansion time variables. The CLEAR macro of Ex. 5.12 is an instance of semantic expansion. Here, the number of `MOVEM  AREG, ..` statements generated by a call on CLEAR is determined by the value of the second parameter of CLEAR. Macro EVAL of example 5.10 is another instance of conditional expansion wherein one of two alternative code sequences is generated depending on the peculiarities of actual parameters of a macro call. Example 5.15 illustrates semantic expansion using the type attribute.

Example 5.15

```
            MACRO
            CREATE.CONST   &X, &Y
            AIF            (T'&X EQ B) .BYTE
&Y          DW             25
            AGO            .OVER
.BYTE       ANOP
&Y          DB             25
.OVER       MEND
```

This macro creates a constant '25' with the name given by the 2nd parameter. The type of the constant matches the type of the first parameter.

EXERCISE 5.4

1. Write a macro that moves 8 numbers from the first 8 positions of an array specified as the first operand into first 8 positions of an array specified as the second operand.
2. Generalize the macro of problem 1 above to move n numbers from the first operand to the second operand, where n is specified as the third operand of the macro.
3. Write a macro which takes A, B, C and D as parameters and calculates A*B + C*D in AREG.
   (a) Where would you store the temporary result?
   (b) Would you reserve space for the temporary result within the macro body or outside it (i.e. in the main program)? Why? What are the advantages and disadvantages of these alternatives?
   (c) Space reserved within the macro would be replicated in every expansion. If this space is named then the name would also be repeated in every expansion. This can lead to duplicate declarations. Can you think of a method to avoid this conflict?
4. Many macro processors permit the operation of concatenation within a macro body to form larger strings. For example, if &C is a parameter with the value M2 then the string XY.&C has the meaning XYM2. (*note:* '.' stands for the concatenation operation.) Could you use this concept to advantage in problem 3(c)?
5. A global EV carries over its value from one macro expansion to another. Can global EV's be used to advantage in problem 3(c)?
6. Study the conditional expansion and SET variable facilities of a macro language accessible to you and code problems 1 to 5 in this macro language.
7. A general purpose macro is to be written to move the contents of one area of memory into another area of memory. Two key issues here are: If the source area is larger in size than the destination area, some of its contents should be ignored (i.e. contents should be *truncated*). If the destination area is larger in size, some part of it should be *padded* with zeroes or blanks. Both truncation and padding can occur at the start or end of the area.

   Code the macro such that user can specify whether truncation and padding should occur at the start or end of the areas.

5.5 DESIGN OF A MACRO PREPROCESSOR

The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions or calls. Figure 5.6 shows a schematic of a macro preprocessor. The program form output by the macro preprocessor can now be handed over to an assembler to obtain the target language form of the program.

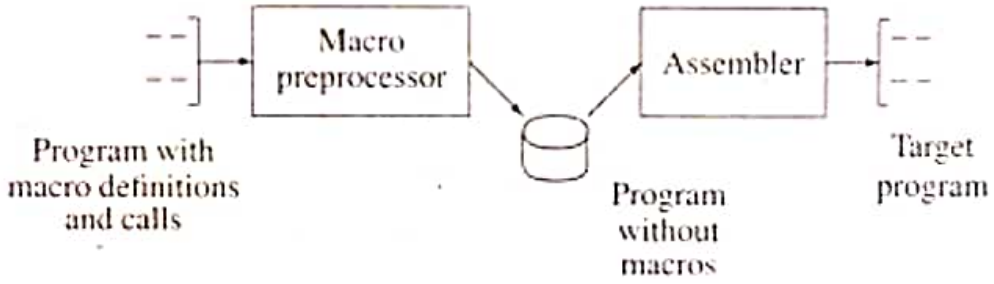


Fig. 5.6 A schematic of a macro preprocessor

Thus the macro preprocessor segregates macro expansion from the process of program assembly. It is economical because it can use an existing assembler. However, it is not as efficient as a macro-assembler, i.e. an assembler that performs macro expansion as well as assembly.

5.5.1 Design Overview

We begin the design by listing all tasks involved in macro expansion.

1. Identify macro calls in the program.
2. Determine the values of formal parameters.
3. Maintain the values of expansion time variables declared in a macro.
4. Organize expansion time control flow.
5. Determine the values of sequencing symbols.
6. Perform expansion of a model statement.

The following 4 step procedure is followed to arrive at a design specification for each task:

1. Identify the information necessary to perform a task.
2. Design a suitable data structure to record the information.
3. Determine the processing necessary to obtain the information.
4. Determine the processing necessary to perform the task.

Application of this procedure to each of the preprocessor tasks is described in the following.

Identify macro calls

A table called the *macro name table* (MNT) is designed to hold the names of all macros defined in a program. A macro name is entered in this table when a macro definition is processed. While processing a statement in the source program, the preprocessor compares the string found in its mnemonic field with the macro names in MNT. A match indicates that the current statement is a macro call.

Determine values of formal parameters

A table called the *actual parameter table* (APT) is designed to hold the values of formal parameters during the expansion of a macro call. Each entry in the table is a pair

(<*formal parameter name*>, <*value*>)

Two items of information are needed to construct this table, names of formal parameters, and default values of keyword parameters. For this purpose, a table called the *parameter default table* (PDT) is used for each macro. This table would be accessible from the MNT entry of a macro and would contain pairs of the form (<*formal parameter name*>, <*default value*>). If a macro call statement does not specify a value for some parameter *par*, its default value would be copied from PDT to APT.

Maintain expansion time variables

An *expansion time variables' table* (EVT) is maintained for this purpose. The table contains pairs of the form

(<*EV name*>, <*value*>)

The value field of a pair is accessed when a preprocessor statement or a model statement under expansion refers to an EV.

Organize expansion time control flow

The body of a macro, i.e. the set of preprocessor statements and model statements in it, is stored in a table called the *macro definition table* (MDT) for use during macro expansion. The flow of control during macro expansion determines when a model statement is to be visited for expansion. Algorithm 5.1 can be used for this purpose. MEC is initialized to the first statement of the macro body in the MDT. It is updated after expanding a model statement or on processing a macro preprocessor statement.

Determine values of sequencing symbols

A *sequencing symbols table* (SST) is maintained to hold this information. The table contains pairs of the form

(<*sequencing symbol name*>, <*MDT entry #*>)

where <*MDT entry #*> is the number of the MDT entry which contains the model statement defining the sequencing symbol. This entry is made on encountering a statement which contains the sequencing symbol in its label field (in the case of a back reference to the symbol), or on encountering a reference prior to its definition (in the case of a forward reference).

Perform expansion of a model statement

This is a trivial task given the following:

1. MEC points to the MDT entry containing the model statement.
2. Values of formal parameters and EV's are available in APT and EVT, respectively.
3. The model statement defining a sequencing symbol can be identified from SST.

Expansion of a model statement is achieved by performing a lexical substitution for the parameters and EV's used in the model statement.

5.5.2 Data Structures

Section 5.5.1 has identified the key data structures of the macro preprocessor. To obtain a detailed design of the data structures it is necessary to apply the practical criteria of processing efficiency and memory requirements

The tables APT, PDT and EVT contain pairs which are searched using the first component of the pair as a key—for example, the formal parameter name is used as the key to obtain its value from APT. This search can be eliminated if the position of an entity within a table is known when its value is to be accessed. We will see this in the context of APT.

The value of a formal parameter ABC is needed while expanding a model statement using it, viz.

```
MOVER      AREG, &ABC
```

Let the pair (ABC, ALPHA) occupy entry #5 in APT. The search in APT can be avoided if the model statement appears as

```
MOVER      AREG, (P,5)
```

in the MDT, where (P, 5) stands for the words 'parameter #5'.

Thus, macro expansion can be made more efficient by storing an intermediate code for a statement, rather than its source form, in the MDT. All parameter names could be replaced by pairs of the form (P, *n*) in model statements and preprocessor statements stored in MDT. An interesting offshoot of this decision is that the

first component of the pairs stored in APT is no longer used during macro expansion, e.g. the information (P, 5) appearing in a model statement is sufficient to access the value of formal parameter ABC. Hence APT containing (*<formal parameter name>*, *<value>*) pairs is replaced by another table called APTAB which only contains *<value>*'s.

To implement this simplification, ordinal numbers are assigned to all parameters of a macro. A table named *parameter name table* (PNTAB) is used for this purpose. PNTAB is used while processing the definition of a macro. Parameter names are entered in PNTAB in the same order in which they appear in the prototype statement. The entry # of a parameter's entry in PNTAB is now its ordinal number. This entry is used to replace the parameter name in the model and preprocessor statements of the macro while storing it in the MDT. This implements the requirement that the statement `MOVER AREG, &ABC` should appear as `MOVER AREG, (P, 5)` in MDT.

In effect, the information (*<formal parameter name>*, *<value>*) in APT has been split into two tables -

PNTAB	which contains formal parameter names
APTAB	which contains formal parameter values (i.e. contains actual parameter)

Note again that PNTAB is used while processing a macro definition while APTAB is used during macro expansion.

Similar analysis leads to splitting of EVT into EVNTAB and EVTAB and SST into SSNTAB and SSTAB. EV names are entered in EVNTAB while processing EV declarations. SS names are entered in SSNTAB while processing an SS reference or definition, whichever occurs earlier.

This arrangement leads to some simplifications concerning PDT. The positional parameters (if any) of a macro appear before keyword parameters in the prototype statement. Hence in the prototype statement for a macro BETA which has p positional parameters and k keyword parameters, the keyword parameters have the ordinal numbers $p+1 \ldots p+k$. Due to this numbering, two kinds of redundancies appear in PDT, the first component of each entry is redundant as in APTAB and EVTAB. Further, entries $1 \ldots p$ are redundant since positional parameters cannot have default specifications. Hence entries only need to exist for parameters numbered $p+1 \ldots p+k$. To accommodate these changes, we replace the parameter default table (PDT) by a *keyword parameter default table* (KPDTAB). KPDTAB of macro BETA would only have k entries in it. To note the mapping that the first entry of KPDTAB corresponds to parameter numbered $p+1$, we store p, the number of positional parameters of macro BETA, in a new field of the MNT entry.

MNT has entries for all macros defined in a program. Each MNT entry contains three pointers MDTP, KPDTP and SSTP, which are pointers to MDT, KPDTAB and SSNTAB for the macro, respectively. Instead of using different MDT's for different macros, for simplicity we can create a single MDT and use different sections of this

Table	*Fields in each entry*
Macro name table (MNT)	*Macro name*, *Number of positional parameters* (#PP), *Number of keyword parameters* (#KP), *Number of expansion time variables* (#EV), *MDT pointer* (MDTP), *KPDTAB pointer* (KPDTP), *SSTAB pointer* (SSTP)
Parameter Name Table (PNTAB)	*Parameter name*
EV Name Table (EVNTAB)	*EV name*
SS Name Table (SSNTAB)	*SS name*
Keyword Parameter Default Table (KPDTAB)	*parameter name*, *default value*
Macro Definition Table (MDT)	*Label*, *Opcode*, *Operands*
Actual Parameter Table (APTAB)	*Value*
EV Table (EVTAB)	*Value*
SS Table (SSTAB)	*MDT entry #*

Fig. 5.7 Tables of the macro preprocessor

table for different macros. A similar arrangement can be used with KPDTAB and SSNTAB. The data structures are now summarized in Fig. 5.7.

Construction and use of the macro preprocessor data structures can be summarized as follows: PNTAB and KPDTAB are constructed by processing the prototype statement. Entries are added to EVNTAB and SSNTAB as EV declarations and SS definitions/references are encountered. MDT entries are constructed while processing the model statements and preprocessor statements in the macro body. An entry is added to SSTAB when the definition of a sequencing symbol is encountered. APTAB

is constructed while processing a macro call. EVTAB is constructed at the start of expansion of a macro.

Example 5.16

```
            MACRO
            CLEARMEM    &X, &N, &REG=AREG
            LCL         &M
&M          SET         0
            MOVER       &REG, ='0'
.MORE       MOVEM       &REG, &X+&M
&M          SET         &M+1
            AIF         (&M NE N) .MORE
            MEND
```

Figure 5.8 shows the contents of the data structures for the call

```
CLEARMEM    AREA, 10
```

Data structures above the broken line are used during the processing of a macro definition, while the data structures between the broken and firm lines are constructed during macro definition processing and used during macro expansion. Data structures below the firm line are used for the expansion of a macro call.

Note how APTAB has been constructed using the actual parameters in the macro call and the default value AREG of the parameter REG. Note also the SSTAB entry for .MORE.

5.5.3 Processing of Macro Definitions

The following initializations are performed before initiating the processing of macro definitions in a program

KPDTAB_pointer := 1;
SSTAB_ptr := 1;
MDT_ptr := 1;

Algorithm 5.2 is invoked for every macro definition in the program.

Algorithm 5.2 (Processing of a macro definition)

1. *SSNTAB_ptr* := 1;
   PNTAB_ptr := 1;
2. Process the macro prototype statement and form the MNT entry
   (a) *name* := macro name;
   (b) For each positional parameter
      (i) Enter *parameter name* in PNTAB [*PNTAB_ptr*].
      (ii) *PNTAB_ptr* := *PNTAB_ptr* + 1;

PNTAB

X
N
REG

EVNTAB

M

SSNTAB

MORE

MNT

name	*#PP*	*#KP*	*#EV*	*MDTP*	*KPDTP*	*SSTP*
CLEARMEM	2	1	1	25	10	5

KPDTAB

10	REG	AREG

SSTAB

5	28

MDT

25		LCL	(E,1)
26	(E,1)	SET	0
27		MOVER	(P,3), ='0'
28		MOVEM	(P,3), (P,1)+(E,1)
29	(E,1)	SET	(E,1)+1
30		AIF	((E,1) NE (P,2)) (S,1)
31		MEND	

APTAB

AREA
10
AREG

EVTAB

0

Fig. 5.8 Data structures of the macro preprocessor

(iii) #PP := #PP + 1;

(c) KPDTP := *KPDTAB_ptr*;

(d) For each keyword parameter

(i) Enter *parameter name* and *default value* (if any), in KPDTAB [*KPDTAB_ptr*].

(ii) Enter *parameter name* in PNTAB [*PNTAB_ptr*].

(iii) *KPDTAB_ptr* := *KPDTAB_ptr* + 1;

(iv) *PNTAB_ptr* := *PNTAB_ptr* + 1;

(v) #KP := #KP + 1;

(e) MDTP := *MDT_ptr*;

(f) #EV := 0;

(g) SSTP := *SSTAB_ptr*;

3. While not a MEND statement

(a) If an LCL statement then

(i) Enter expansion time variable name in EVNTAB.

(ii) #EV := #EV + 1;

(b) If a model statement then

(i) If label field contains a sequencing symbol then

If symbol is present in SSNTAB then

q := entry number in SSNTAB;

else

Enter symbol in SSNTAB [*SSNTAB_ptr*].

q := *SSNTAB_ptr*;

SSNTAB_ptr := *SSNTAB_ptr* + 1;

SSTAB [SSTP + q − 1] := *MDT_ptr*;

(ii) For a parameter, generate the specification (P, #*n*).

(iii) For an expansion variable, generate the specification (E, #*m*).

(iv) Record the IC in MDT [*MDT_ptr*];

(v) *MDT_ptr* := *MDT_ptr* + 1;

(c) If a preprocessor statement then

(i) If a SET statement

Search each expansion time variable name used in the statement i
EVNTAB and generate the spec (E, #*m*).

(ii) If an AIF or AGO statement then

If sequencing symbol used in the statement is present in SSNTAI
then

q := entry number in SSNTAB;

(iii) #PP := #PP + 1;

(c) KPDTP := *KPDTAB_ptr*;

(d) For each keyword parameter

(i) Enter *parameter name* and *default value* (if any), in KPDTAB [*KPDTAB_ptr*].

(ii) Enter *parameter name* in PNTAB [*PNTAB_ptr*].

(iii) *KPDTAB_ptr* := *KPDTAB_ptr* + 1;

(iv) *PNTAB_ptr* := *PNTAB_ptr* + 1;

(v) #KP := #KP + 1;

(e) MDTP := *MDT_ptr*;

(f) #EV := 0;

(g) SSTP := *SSTAB_ptr*;

3. While not a MEND statement

(a) If an LCL statement then

(i) Enter expansion time variable name in EVNTAB.

(ii) #EV := #EV + 1;

(b) If a model statement then

(i) If label field contains a sequencing symbol then

If symbol is present in SSNTAB then

q := entry number in SSNTAB;

else

Enter symbol in SSNTAB [*SSNTAB_ptr*].

q := *SSNTAB_ptr*;

SSNTAB_ptr := *SSNTAB_ptr* + 1;

SSTAB [SSTP + q − 1] := *MDT_ptr*;

(ii) For a parameter, generate the specification (P, #n).

(iii) For an expansion variable, generate the specification (E, #m).

(iv) Record the IC in MDT [*MDT_ptr*];

(v) *MDT_ptr* := *MDT_ptr* + 1;

(c) If a preprocessor statement then

(i) If a SET statement

Search each expansion time variable name used in the statement i
EVNTAB and generate the spec (E, #m).

(ii) If an AIF or AGO statement then

If sequencing symbol used in the statement is present in SSNTA
then

q := entry number in SSNTAB;

else
Enter symbol in SSNTAB [*SSNTAB_ptr*].
q := *SSNTAB_ptr*;
SSNTAB_ptr := *SSNTAB_ptr* + 1;
Replace the symbol by (S, SSTP + q − 1).

(iii) Record the IC in MDT [*MDT_ptr*].

(iv) *MDT_ptr* := *MDT_ptr* + 1;

4. (MEND statement)
If *SSNTAB_ptr* = 1 (i.e. SSNTAB is empty) then
SSTP := 0;
Else *SSTAB_ptr* := *SSTAB_ptr* + *SSNTAB_ptr* − 1;
If #KP = 0 then KPDTP = 0;

5.5.4 Macro Expansion

We use the following data structures to perform macro expansion:

APTAB	Actual parameter table
EVTAB	EV table
MEC	Macro expansion counter
APTAB_ptr	APTAB pointer
EVTAB_ptr	EVTAB pointer

The number of entries in APTAB (i.e., $\#e_{APTAB}$) equals the sum of values in the #PP and #KP fields of the MNT entry of a macro. Number of entries in EVTAB (i.e., $\#e_{EVTAB}$) is given by the value in #EV field of the MNT. APTAB and EVTAB are constructed when a macro call is recognized. *APTAB_ptr* and *EVTAB_ptr* are set to point at these tables. As in Algorithm 5.1, MEC always points to the next statement to be expanded.

Algorithm 5.3 (Macro expansion)

1. Perform initializations for the expansion of a macro
   (a) MEC := MDTP field of the MNT entry;
   (b) Create EVTAB with #EV entries and set *EVTAB_ptr*.
   (c) Create APTAB with #PP+#KP entries and set *APTAB_ptr*.
   (d) Copy keyword parameter defaults from the entries KPDTAB [KPDTP] ... KPDTAB [KPDTP+#KP−1] into APTAB [#PP+1] ... APTAB [#PP+#KP].
   (e) Process positional paremeters in the actual parameter list and copy them into APTAB [1] ... APTAB [#PP].

(f) For keyword parameters in the actual parameter list
Search the keyword name in *parameter name* field of KPDTAB [KPDTP] ... KPDTAB [KPDTP+#KP−1]. Let KPTDAB [*q*] contain a matching entry. Enter value of the keyword parameter in the call (if any) in APTAB [#PP+*q*−KPDTP+1].

2. While statement pointed by MEC is not MEND statement
   (a) If a model statement then
      (i) Replace operands of the form (P, #*n*) and (E, #*m*) by values in APTAB [*n*] and EVTAB [*m*] respectively.
      (ii) Output the generated statement.
      (iii) MEC := MEC + 1;
   (b) If a SET statement with the specification (E, #*m*) in the label field then
      (i) Evaluate the expression in the operand field and set an appropriate value in EVTAB [*m*].
      (ii) MEC := MEC + 1;
   (c) If an AGO statement with (S, #*s*) in operand field then
      MEC := SSTAB [SSTP+*s* − 1];
   (d) If an AIF statement with (S, #*s*) in operand field then
      If condition in the AIF statement is *true* then
         MEC := SSTAB [SSTP+*s* − 1];
3. Exit from macro expansion.

Figure 5.9 shows the data structures at some point during the expansion of macro CLEARMEM of Ex. 5.16.

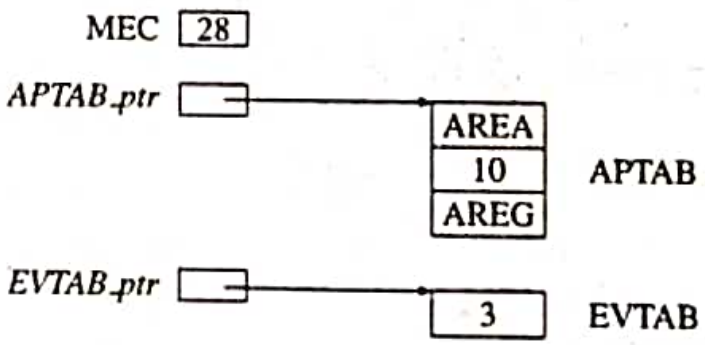


Fig. 5.9 Data structures during macro expansion

5.5.5 Nested Macro Calls

Figure 5.5 in Section 5.3 illustrates nested macro calls. Two basic alternatives exist for processing nested macro calls. We can apply the macro expansion schematic of

Section 5.5.4 to obtain the first level expanded code, viz.

```
                - - -
+               MOVEM       BREG, TMP
+               INCR_D      X, Y, REG=BREG
+               MOVER       BREG, TMP
                - - -
```

In this code macro calls appearing in the source program have been expanded but statements resulting from the expansion may themselves contain macro calls. The macro expansion schematic can be applied to the first level expanded code to expand these macro calls and so on, until we obtain a code form which does not contain any macro calls. This scheme would require a number of passes of macro expansion, which makes it quite expensive.

A more efficient alternative would be to examine each statement generated during macro expansion to see if it is itself a macro call. If so, a provision can be made to expand this call before continuing with the expansion of the parent macro call. This avoids multiple passes of macro expansion, thus ensuring processing efficiency. This alternative requires some extensions in the macro expansion scheme of Section 5.5.4.

Consider the situation when the ADD statement marked $\boxed{3}$ in Fig. 5.5 is being generated. Expansion of two macro calls is in progress at this moment. This happened because the outer macro COMPUTE gave rise to a macro call on macro INCR_D during the expansion of its current model statement. The model statements of INCR_D are currently being expanded using the expansion time data structures MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr*. A MEND statement encountered during the expansion must lead to resumption of expansion of the outer macro. This requires that MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr* should be restored to the values contained in them while the macro COMPUTE was being expanded. Control returns to the processing of the source program when the MEND statement is encountered during the processing of COMPUTE.

Thus, two provisions are required to implement the expansion of nested macro calls:

1. Each macro under expansion must have its own set of data structures, viz. MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr*.
2. An *expansion nesting counter* (*Nest_cntr*) is maintained to count the number of nested macro calls. *Nest_cntr* is incremented when a macro call is recognized and decremented when a MEND statement is encountered. Thus *Nest_cntr* > 1 indicates that a nested macro call is under expansion, while *Nest_cntr* = 0 implies that macro expansion is not in progress currently.

The first provision can be implemented by creating many copies of the expansion time data structures. These can be stored in the form of an array. For example, we can

have an array called APTAB_ARRAY, each element of which is an APTAB. APTAB for the innermost macro call would be given by APTAB_ARRAY[*Nest_cntr*]. This arrangement provides access efficiency. However, it is expensive in terms of memory requirements. It also involves a difficult design decision—how many copies of the data structures should be created? If too many copies are created, some may never be used. If too few are created, some assembly programs may have to be rejected because their macro call nesting depth exceeds the number of copies of the data structures.

Since macro calls are expanded in a LIFO manner, a practical solution is to use a stack to accommodate the expansion time data structures. The stack consists of *expansion records*, each expansion record accommodating one set of expansion time data structures. The expansion record at the top of the stack corresponds to the macro call currently being expanded. When a nested macro call is recognized, a new expansion record is pushed on the stack to hold the data structures for the call. At MEND, an expansion record is popped off the stack. This would uncover the previous expansion record in the stack which houses the expansion time data structures of the outer macro. The extended stack model of Section 2.2.1 is used for this purpose.

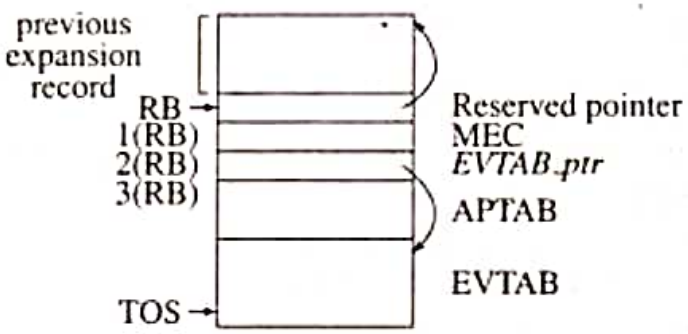


Fig. 5.10 Use of stack for macro preprocessor data structures

Figure 5.10 illustrates the stack oriented management of expansion time data structures. The expansion record on top of the stack contains the data structures in current use. *Record base* (RB) is a pointer pointing to the start of this expansion record. TOS points to the last occupied entry in stack. When a nested macro call is detected, another set of data structures is allocated on the stack. RB is now set to point to the start of the new expansion record. MEC, *EVTAB_ptr*, APTAB and EVTAB are allocated on the stack in that order. During macro expansion, the various data structures are accessed with reference to the value contained in RB. This is performed using the following addresses:

Data structure	*Address*
Reserved pointer	0(RB)
MEC	1(RB)

EVTAB_ptr	2(RB)
APTAB	3(RB) to e_{APTAB}+2(RB)
EVTAB	contents of *EVTAB_ptr*

where 1(RB) stands for 'contents of RB+1'. Note that the first entry of APTAB always has the address 3(RB). This eliminates the need for *APTAB_ptr*.

At a MEND statement, a record is popped off the stack by setting TOS to the end of the previous record. It is now necessary to set RB to point to the start of previous record in stack. This is achieved by using the entry marked 'reserved pointer' in the expansion record. This entry always points to the start of the previous expansion record in stack. While popping off a record, the value contained in this entry can be loaded into RB. This has the effect of restoring access to the expansion time data structures used by the outer macro.

Actions at start and end of a macro expansion are based on the extended stack model of Section 2.2.1.

At start of expansion

Actions at the start of expansion are summarized in Table 5.1.

Table 5.1 Actions at start of macro expansion

No.			*Statement*
1.	TOS	:=	TOS+1;
2.	TOS*	:=	RB;
3.	RB	:=	TOS;
4.	1(RB)	:=	MDTP entry of MNT;
5.	2(RB)	:=	RB+3+#e_{APTAB};
6.	TOS	:=	TOS + #e_{APTAB} + #e_{EVTAB}+2;

The first statement increments TOS to point at the first word of the new expansion record. This is the *reserved pointer*. The '*' mark in the second statement TOS* := RB indicates indirection. This statement deposits the address of the previous record base into this word. New RB is now established in statement 3. Statements 4 and 5 set MEC and *EVTAB_ptr* respectively. Statement 6 sets TOS to point to the last entry of the expansion record.

At end of expansion

Actions at the end of expansion are summarized in Table 5.2.

The first statement pops an expansion record off the stack by resetting TOS to the value it had while the outer macro was being expanded. RB is then made to point at the base of the previous record. Data structures in the old expansion record are now accessible as displacements from the new value in RB, e.g. MEC is 1(RB).

Table 5.2 Actions at end of macro expansion

No.	*Statement*
1.	TOS := RB−1;
2.	RB := RB*;

Example 5.17 Figure 5.11 illustrates the stack during the expansion of nested macro calls of Fig. 5.4. The statement `INCR_D &FIRST, &SECOND` of macro COMPUTE is assumed to occupy entry 18 in MDT. The body of macro INCR_D is assumed to occupy entries 73-75. Parts (a), (b) and (c) of the figure show the stack before, during and after the expansion of the nested macro call `INCR_D X,Y`. In part (a) of the figure, RB points at the bottom-most entry of the stack and TOS points at the last entry of the expansion record, i.e. at the entry of Y in APTAB. When the nested call is encountered, the *reserved pointer* of the new record would be made to point at the start of the previous expansion record. MEC and APTAB for the call would now be established.

After completing the expansion of the call on INCR_D, RB would be reset to the bottom-most entry of the stack, and TOS would once again point at the entry of Y in APTAB of the first record. MEC now contains the value 18, which would be incremented to 19. Macro expansion thus continues with the next statement in macro COMPUTE.

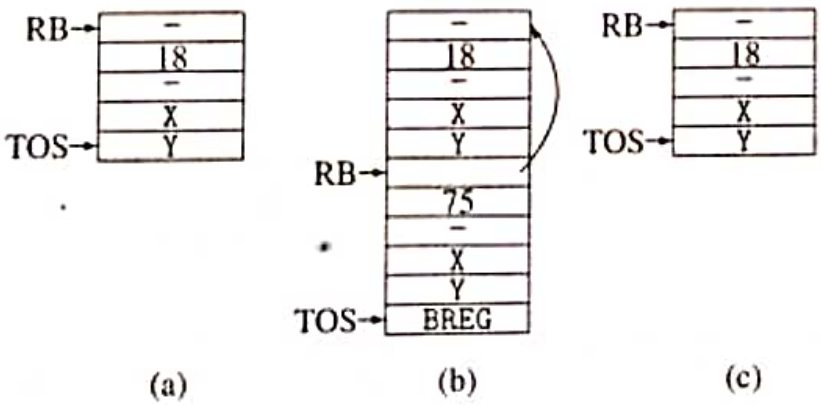


Fig. 5.11 An illustration of the expansion of nested macro calls

EXERCISE 5.5

1. Modify Algorithm 5.3 to enable expansion of nested macro calls.

5.5.6 Design of a Macro-assembler

As mentioned in Section 5.5, use of a macro preprocessor followed by a conventional assembler (see Fig. 5.6) is an expensive way of handling macros since the number of passes over the source program is large and many functions get duplicated. For example, analysis of a source statement to detect macro calls requires us to process

the mnemonic field. A similar function is required in the first pass of the assembler. Similar functions of the preprocessor and the assembler can be merged if macros are handled by a macro assembler which performs macro expansion and program assembly simultaneously. This may also reduce the number of passes.

The discussion in the previous section may give rise to the impression that it is always possible to perform macro expansion in a single pass. This is not true, as certain kinds of forward references in macros cannot be handled in a single pass.

Example 5.18 Consider the assembly program of Fig. 5.12. Here the type attribute `T'&X` in macro CREATE_CONST is a forward reference to symbol `A`. Processing of the type attribute cannot be postponed because it determines the nature of the constant defined by the macro. In turn, this affects memory allocation and hence the address of `A` !

```
            MACRO
            CREATE_CONST   &X, &Y
            AIF            (T'&X EQ B) .BYTE
&Y          DW             25
            AGO            .OVER
.BYTE       ANOP
&Y          DB             25
.OVER       MEND

            CREATE_CONST   A, NEW_CON
            :
A           DB             ?
            END
```

Fig. 5.12 Forward reference in type attribute of a parameter

This problem leads to the classical two pass organization for macro expansion. The first pass collects information about the symbols defined in a program and the second pass performs macro expansion.

Pass structure of a macro-assembler

To design the pass structure of a macro-assembler we identify the functions of a macro preprocessor and the conventional assembler which can be merged to advantage. After merging, the functions can be structured into passes of the macro-assembler. This process leads to the following pass structure:

Pass I

1. Macro definition processing
2. SYMTAB construction.

Pass II

1. Macro expansion

2. Memory allocation and LC processing
3. Processing of literals
4. Intermediate code generation.

Pass III

1. Target code generation.

Pass II is large in size since it performs many functions. Further, since it performs macro expansion as well as Pass I of a conventional assembler, all the data structures of the macro preprocessor and the conventional assembler need to exist during this pass.

The pass structure can be simplified if attributes of actual parameters are not to be supported. The macro preprocessor would then be a single pass program. Integrating Pass I of the assembler with the preprocessor would give us the following two pass structure:

Pass I

1. Macro definition processing
2. Macro expansion
3. Memory allocation, LC processing and SYMTAB construction
4. Processing of literals
5. Intermediate code generation.

Pass II

1. Target code generation.

There is obvious imbalance between the sizes of the two passes. A 3-pass structure might be preferred for this reason alone.

EXERCISE 5.5

1. Compare and contrast the properties of macros and subroutines with respect to the following:
   (a) code space requirements
   (b) execution speed
   (c) processing required by the assembler
   (d) flexibility and generality.
2. In an assembly language program, a certain action is required at 10 places in the program. Under what conditions would you code this action as
   (a) a macro?
   (b) a subroutine?

   Justify your answer with the help of appropriate examples.
3. Solve the problems of exercise 5.4.3 using REPT and IRP statements.

4. Extend the macro preprocessor described in this chapter to support the following features:
   (a) REPT and IRP statements discussed in section 5.4.2
   (b) Global EV's
   (c) Nested macro calls.

BIBLIOGRAPHY

Flores (1971), Donovan (1972) and Cole (1981) are few of the texts covering macro processors and macro assemblers in detail. Macros have been used as a tool for writing portable programs. Brown (1974) and Wallis (1982) discuss these aspects in detail.

1. Brown, P.J. (1974): *Macro Processors and Techniques for Portable Software*, Wiley, London.
2. Cole, A.J. (1981): *Macro Processors*, Cambridge University Press, Cambridge.
3. Donovan, J.J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
4. Flores, I. (1971): *Assemblers and BAL*, Prentice-Hall, Englewood Cliffs.
5. Wallis, P.J.L. (1982): *Portable Programming*, Macmillan, London, 1982.

CHAPTER 6

Compilers and Interpreters

6.1 ASPECTS OF COMPILATION

A compiler bridges the semantic gap between a PL domain and an execution domain. Two aspects of compilation are:

1. Generate code to implement meaning of a source program in the execution domain.
2. Provide diagnostics for violations of PL semantics in a source program.

To understand the issues involved in implementing these aspects, we briefly discuss PL features which contribute to the semantic gap between a PL domain and an execution domain. These are:

1. Data types
2. Data structures
3. Scope rules
4. Control structure.

Data types

Definition 6.1 (Data type) *A* data type *is the specification of* (*i*) *legal values for variables of the type, and* (*ii*) *legal operations on the legal values of the type.*

Legal operations of a type typically include an assignment operation and a set of data manipulation operations. Semantics of a data type require a compiler to ensure that variables of a type are assigned or manipulated only through legal operations. The following tasks are involved in ensuring this:

1. Checking legality of an operation for the types of its operands. This ensures that a variable is subjected only to legal operations of its type.
2. Use type conversion operations to convert values of one type into values of another type wherever necessary and permissible according to the rules of a PL.
3. Use appropriate instruction sequences of the target machine to implement the operations of a type.

Example 6.1 Consider the Pascal program segment

```
var
   x, y : real;
   i, j : integer;
begin
   y := 10;
   x := y + i;
```

While compiling the first assignment statement, the compiler must note that y is a real variable, hence every value stored in y must be a real number. Therefore it must generate code to convert the value '10' to the floating point representation. In the second assignment statement, the addition cannot be performed on the values of y and i straightaway as they belong to different types. Hence the compiler must first generate code to convert the value of i to the floating point representation and then generate code to perform the addition as a floating point operation.

Having checked the legality of each operation and determined the need for type conversion operations, the compiler must generate *type specific code* to implement an operation. In a type specific code the value of a variable of type $type_i$ is always manipulated through instructions which know how values of $type_i$ are represented.

Example 6.2 In example 1.12, we have seen how the instructions

```
CONV_R      AREG, I
ADD_R       AREG, B
MOVEM       AREG, A
```

are generated for the program segment

```
i : integer;
a,b : real;
a := b+i;
```

Each instruction is type specific, i.e. it expects the operand to be of a specific type.

Generation of type specific code achieves two important things. It implements the second half of a type's definition, viz. a value of $type_i$ is only manipulated through a legal operation of $type_i$. It also ensures execution efficiency since type related issues do not need explicit handling in the execution domain.

Data structures

A PL permits the declaration and use of data structures like arrays, stacks, records, lists, etc. To compile a reference to an element of a data structure, the compiler must develop a memory mapping to access the memory word(s) allocated to the element. A record, which is a heterogeneous data structure, leads to complex memory mappings. A user defined type requires mappings of a different kind—those that map the values of the type into their representations in a computer, and vice versa.

Example 6.3 Consider the Pascal program segment

```
program example (input, output);
   type
      employee = record
         name : array [1..10] of character;
         sex : character;
         id : integer
      end;
      weekday = (mon, tue, wed, thu, fri);
   var
      info : array [1..500] of employee;
      today : weekday;
      i,j : integer;
   begin { Main program }
         today := mon;
         info[i].id := j;
         if today = tue then ..
end.
```

Here, `info` is an array of records. The reference `info[i].id` involves use of two different kinds of mappings. The first one is the homogeneous mapping of an array reference. This is used to access `info[i]`. The second mapping is used to access the field `id` within an element of `info`, which is a data item of type `employee`. `weekday` is a user defined data type. The compiler must first decide how to represent different values of the type, and then develop an appropriate mapping between the values `mon .. fri` and their representations. A popular technique is to map these values into a subrange of integers. For example, `mon .. fri` can be mapped into the subrange 1 .. 5. This does not lead to confusion between these values and integers because the legality check is applied to each operation. Thus `tue+10` is an illegal expression even if `tue` is represented by the value '2', because '+' is not a legal operation of type `weekday`.

Scope rules

Scope rules determine the accessibility of variables declared in different blocks of a program. The *scope* of a program entity (e.g. a data item) is that part of a program where the entity is accessible. In most languages the scope of a data item is restricted to the program block in which the data item is declared. It extends to an enclosed block unless the enclosed block declares a variable with an identical name.

Example 6.4

```
A [ x,y : real;
    B [ y,z : integer;
        x := y;
    ]
]
```

Variable x of block A is accessible in block A and in the enclosed block B. However, variable y of block A is not accessible in block B since y is redeclared in block B. Thus, the statement `x := y` uses y of block B.

The compiler performs operations called *scope analysis* and *name resolution* to determine the data item designated by the use of a name in the source program. The generated code simply implements the results of the analysis. Section 6.2.2 contains a detailed discussion of the implementation of scope rules.

Control structure

The *control structure* of a language is the collection of language features for altering the flow of control during the execution of a program. This includes conditional transfer of control, conditional execution, iteration control and procedure calls. The compiler must ensure that a source program does not violate the semantics of control structures.

Example 6.5 In the Pascal program segment

```
for i := 1 to 100 do
begin
   lab1: if i = 10 then ..
end;
```

a control transfer to the statement bearing label `lab1` from outside the loop is forbidden. Some languages (though not Pascal !) also forbid assignments to the control variable of a **for** loop within the body of the loop.

6.2 MEMORY ALLOCATION

Memory allocation involves three important tasks

1. Determine the amount of memory required to represent the value of a data item.
2. Use an appropriate memory allocation model to implement the lifetimes and scopes of data items.
3. Determine appropriate memory mappings to access the values in a non scalar data item, e.g. values in an array.

The first task is implemented during semantic analysis of data declaration statements. In this section, we discuss the static and dynamic memory allocation models, and memory mappings used in the case of arrays. Discussion of advanced memory mappings to handle data alignment requirements in vector data, e.g. records, can be found in the literature cited at the end of the chapter.

6.2.1 Static and Dynamic Memory Allocation

Following Definition 1.8, we can define memory binding as follows:

Definition 6.2 (Memory binding) *A* memory binding *is an association between the 'memory address' attribute of a data item and the address of a memory area.*

Memory allocation is the procedure used to perform memory binding. The binding ceases to exist when memory is deallocated. Memory bindings can be static or dynamic in nature (see Definitions 1.9 and 1.10), giving rise to the static and dynamic memory allocation models. In *static memory allocation*, memory is allocated to a variable *before* the execution of a program begins. Static memory allocation is typically performed during compilation. No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated; allocation to a variable exists even if the program unit in which it is defined is not active. In *dynamic memory allocation*, memory bindings are established and destroyed *during* the execution of a program. Typical examples of the use of these memory allocation models are Fortran for static allocation and block structured languages like PL/I, Pascal, Ada, etc., for dynamic allocation.

Example 6.6 Figure 6.1 illustrates static and dynamic memory allocation to a program consisting of 3 program units—A, B and C. Part (a) shows static memory allocation. Part (b) shows dynamic allocation when only program unit A is active. Part (c) shows the situation after A calls B, while Part (d) shows the situation after B returns to A and A calls C. C has been allocated part of the memory deallocated from B. It is clear that static memory allocation allocates more memory than dynamic memory allocation except when all program units are active.

Dynamic memory allocation has two flavours—automatic allocation and program controlled allocation. According to the terminology of Section 1.4.2, the former implies memory binding performed at execution init time of a program unit, while the latter implies memory binding performed during the execution of a program unit. We describe the details of these bindings in the following.

In *automatic dynamic allocation*, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exited. Thus the same memory area may be used for the variables of different program units (see Fig. 6.1). It is also possible that different memory areas may be allocated to the same variable in different activations of a program unit, e.g. when some procedure is invoked in different blocks of a program. In

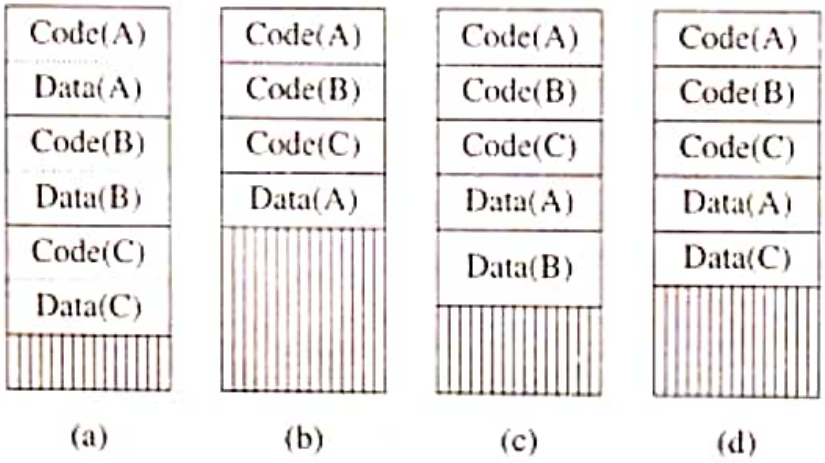


Fig. 6.1 Static and dynamic memory allocation

program controlled dynamic allocation, a program can allocate or deallocate memory at arbitrary points during its execution. It is obvious that in both automatic and program controlled allocation, address of the memory area allocated to a program unit cannot be determined at compilation time.

Dynamic memory allocation is implemented using stacks and heaps, thus necessitating pointer based access to variables. This tends to make it slower in execution than static memory allocation. Automatic dynamic allocation is implemented using a stack since entry and exit from program units is LIFO in nature (see Section 6.2.2 for more details). When a program unit is entered during the execution of a program, a record is created in the stack to contain its variables. A pointer is set to point to this record. Individual variables of the program unit are accessed using displacements from this pointer. Program controlled dynamic allocation is implemented using a heap. A pointer is now needed to point to each allocated memory area.

Example 6.7 Let program unit A contain a declaration for a simple variable `alpha`. When static memory allocation is used the compiler can use the absolute address of `alpha` to implement accesses to it. When dynamic allocation is used, let the compiler maintain the address of the memory record created for A in some register r. Address of variable `alpha` is now

$$< r > + d_{\texttt{alpha}}$$

where $d_{\texttt{alpha}}$ is the displacement of the memory area allocated to `alpha` from the start of the record for A. The constant $d_{\texttt{alpha}}$ is determined during compilation.

Dynamic allocation provides some significant advantages. Recursion can be implemented easily because memory is allocated when a program unit is entered during execution. This leads to allocation of a separate memory area for each recursive activation of a program unit. (We discuss this aspect later in the section.) Dynamic allocation can also support data structures whose sizes are determined dynamically, e.g. an array declaration `a [m,n]`, where `m` and `n` are variables.

6.2.2 Memory Allocation in Block Structured Languages

A *block* is a program unit which can contain data declarations. A program in a block structured language is a nested structure of blocks. A block structured language uses dynamic memory allocation. Algol-60 is the first widely used block structured language. The concept of block structure is also used in PL/I, Algol-68, Pascal, Ada and many other languages.

Scope rules

A data declaration using a name $name_i$ creates a variable var_i and establishes a binding between $name_i$ and var_i. We will represent this binding as $(name_i, var_i)$, and call it the name–var binding. Variable var_i is *visible* at a place in the program if some binding $(name_i, var_i)$ is effective at that place. A visible variable can be accessed using its name, $name_i$ in the above case. It is possible for data declarations in many blocks of a program to use a same name, say, $name_i$. This would establish many bindings of the form $(name_i, var_k)$ for different values of k. Scope rules determine which of these bindings is effective at a specific place in the program.

Consider a block b which contains a data declaration using the name $name_i$. This establishes a binding $(name_i, var_i)$ for a variable var_i. This binding only exists within block b. If a block b' nested inside block b also contains a declaration using $name_i$, a binding $(name_i, var_j)$ exists in b' for some variable var_j. This suppresses the binding $(name_i, var_i)$ over block b'. Thus variable var_i is not visible in b'. However, if b' did not contain a declaration using $name_i$, the binding $(name_i, var_i)$ would have been effective over b' as well. Thus, var_i would have been visible in b'. We summarize the rules governing visibility of a variable in the following.

Scope of a variable

If a variable var_i is created with the name $name_i$ in a block b,

1. var_i can be accessed in any statement situated in block b.
2. var_i can be accessed in any statement situated a block b' which is enclosed in b, unless b' contains a declaration using the same name (i.e. using the name $name_i$).

A variable declared in block b is called a *local variable* of block b. A variable of an enclosing block, that is accessible within block b, is called a *nonlocal variable* of block b. The following notation is used to differentiate between variables created using the same name in different blocks:

$name_{block_name}$: variable created by a data declaration using the name *name* in block *block_name*

Thus $\texttt{alpha}_A$, $\texttt{alpha}_B$ are variables created using the name `alpha` in blocks A and B.

Example 6.8 Consider the block-structured program in Fig. 6.2. The variables accessible within the various blocks are as follows:

```
A [ x,y,z : integer;
    B [ g : real;
        C [ h,z : real;
          ]
    D [ i,j : integer;
      ]
  ]
```

Fig. 6.2 Block structured program

Block	*Accessible variables*	
	local	*nonlocal*
A	x_A, y_A, z_A	–
B	g_B	$x_A, y_A\ z_A$
C	h_C, z_C	x_A, y_A, g_B
D	i_D, j_D	x_A, y_A, z_A

Variable z_A is not accessible inside block C since C contains a declaration using the name z. Thus z_A and z_C are two distinct variables. This would be true even if they had identical attributes, i.e. even if z of C was declared to be an integer.

Memory allocation and access

Automatic dynamic allocation is implemented using the extended stack model (see Section 2.2.1) with a minor variation—each record in the stack has *two* reserved pointers instead of one (see Fig. 6.3). Each stack record accommodates the variables for one activation of a block, hence we call it an *activation record* (AR). The following notation is used to refer to the activation record of a block:

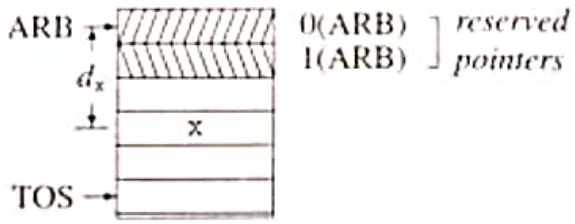


Fig. 6.3 Stack record format

AR_A^i : Activation record for the ith activation of A

wherein we omit the superscript i unless multiple activations of a block exist. During the execution of a block structured program, a register called the *activation record base* (ARB) always points to the start address of the TOS record. This record belongs to the block which contains the statement being executed. A local variable x of this block is accessed using the address d_x(ARB), where d_x is the displacement of variable x from the start of AR. The address may also be written as <ARB> + d_x, where <ARB> stands for the words 'contents of ARB'.

Dynamic pointer

The first reserved pointer in a block's AR points to the activation record of its dynamic parent. This is called the *dynamic pointer* and has the address 0(ARB). The dynamic pointer is used for deallocating an AR. Actions at block entry and exit are analogous to those described in Section 2.2.1.

Example 6.9 Figure 6.4 depicts memory allocation for the following program:

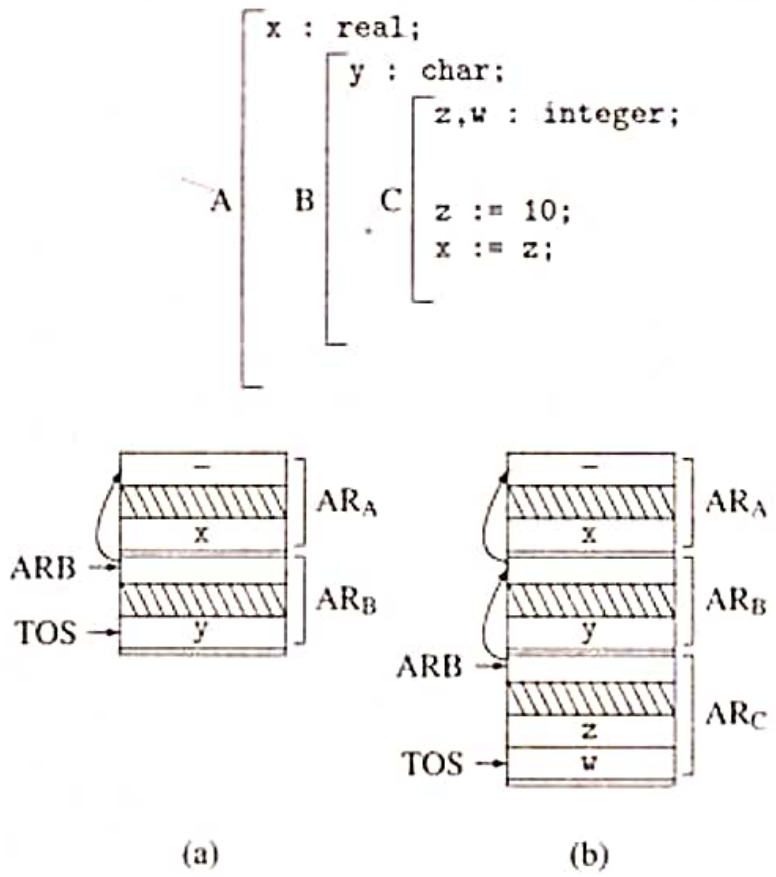


Fig. 6.4 Dynamic allocation and access

where each block could be a **begin** block in Algol, PL/I or a procedure without parameters in some block structured language. It is assumed that the blocks are entered in the sequence A, B, C during the execution of the program.

Figure 6.4(a) shows the situation when blocks A and B are active. Figure 6.4(b) shows the situation after entry to C. Situation after exit from C would again be as shown in

Fig. 6.4(a).

Actions at entry

The actions at entry of block C are described in Table 6.1.

Table 6.1 Actions at block entry

No.	*Statement*
1.	TOS := TOS + 1;
2.	TOS* := ARB; {Set the dynamic pointer}
3.	ARB := TOS;
4.	TOS := TOS + 1;
5.	TOS* := ...; {Set reserved pointer 2}
6.	TOS := TOS + n;

where n is the number of memory words allocated to variables of block C, i.e. n = 2. After step 1, TOS points to the dynamic pointer of the AR being created. Step 2 sets the dynamic pointer to point to the previous activation record. Step 3 sets ARB to the start of the new AR. Step 6 is the memory allocation step. Note that positions of local variables z and w in AR_C are determined during compilation using their type and length information. d_z and d_w are 2 and 3 assuming an integer to occupy 1 word. Thus the address of variable z is <ARB> + 2.

Actions at exit

The actions at entry of block C are described in Table 6.2.

Table 6.2 Actions at block exit

No.	*Statement*
1.	TOS := ARB − 1;
2.	ARB := ARB*;

The first statement sets TOS to point to the last word of the previous activation record. The second statement sets ARB to point to the start of the previous activation record. These actions effectively deallocate the TOS AR in the stack.

Accessing nonlocal variables

A nonlocal variable *nl_var* of a block *b_use* is a local variable of some block *b_defn* enclosing *b_use* (trivially *b_defn* and *b_use* could be the same block). The following terminology is used for a block and its enclosing blocks. A *textual ancestor* or *static ancestor* of block *b_use* is a block which encloses block *b_use*. The block immediately enclosing *b_use* is called its Level 1 ancestor. A Level m ancestor is a block

which immediately encloses the Level $(m-1)$ ancestor. The *level difference* between *b_use* and its Level m ancestor is m. If $s_nest_{b_use}$ represents the static nesting level of block *b_use* in the program, *b_use* has a Level i ancestor, $\forall i < s_nest_{b_use}$.

According to the rules of a block structured language, when *b_use* is in execution, *b_defn* must be active. Hence AR_{b_defn} exists in the stack, and *nl_var* is to be accessed as

$$\text{start address of } AR_{b_defn} + d_{nl_var}$$

where d_{nl_var} is the displacement of *nl_var* in AR_{b_defn}.

Static pointer

Access to nonlocal variables is implemented using the second reserved pointer in AR. This pointer, which has the address 1(ARB), is called the *static pointer*. When an AR is created for a block b, its static pointer is set to point to the AR of the static ancestor of b. The code to access a nonlocal variable *nl_var* declared in a Level m ancestor of *b_use*, $m \geq 1$, is as follows:

1. r := ARB; where r is some register.
2. Repeat step 3 m times.
3. $r := 1(r)$; i.e. load the static pointer into register r.
4. Access *nl_var* using the address $< r > + d_{nl_var}$. (6.1)

Thus a nonlocal variable defined in a Level m ancestor is accessed using m indirections through the static pointer. Note that the value of m is known during compilation from the static structure of the program.

Example 6.10 Figure 6.5 shows the status of memory allocation after executing statement `z := 10;` of block `C` in the program of Ex. 6.9. Steps 4 and 5 of the actions at block entry are:

No.	*Statement*
4.	TOS := TOS + 1;
5.	TOS* := address of AR of Level 1 ancestor.

When block `C` is entered, these actions set the static pointer in AR_C to point at the start of AR_B. When `B` was entered, the static pointer in AR_B would have been set to point at the start of AR_A. The code generated to access variable `x` in the statement `x := z;` is

1. r := ARB;
2. $r := 1(r)$;
3. $r := 1(r)$;
4. Access `x` using the address $< r > + d_x$.

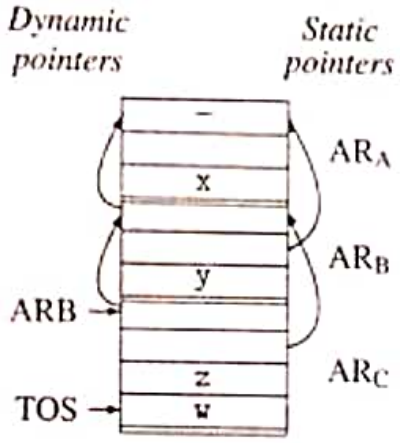


Fig. 6.5 Accessing nonlocal variables with static pointers

Displays

For large values of level difference, it is expensive to access nonlocal variables using static pointers. *Display* is an array used to improve the efficiency of nonlocal accesses. When a block B is in execution, the entries of *Display* contain the following information:

$$\begin{aligned}
\text{Display}[1] &= \text{address of Level } (s_nest_b - 1) \text{ ancestor of B.} \\
\text{Display}[2] &= \text{address of Level } (s_nest_b - 2) \text{ ancestor of B.} \\
&\vdots \\
\text{Display}[s_nest_b - 1] &= \text{address of Level 1 ancestor of B.} \\
\text{Display}[s_nest_b] &= \text{address of } AR_B.
\end{aligned}$$

Let block B refer to some variable v_j defined in an ancestor block b_i. The address of v_j is calculated as

$$\text{Display}[s_nest_{b_i}] + d_{v_j}$$

Hence the code generated for the access would be

1. r := Display[$s_nest_{b_i}$];
2. Access v_j using the address $< r > + d_{v_j}$; (6.2)

which is more efficient compared with the steps in (6.1).

Example 6.11 Figure 6.6 illustrates the contents of *Display* while executing the program of Ex. 6.9. The reference to variable x in block C is implemented using the code

1. r := Display[1];
2. Access x using the address $< r > + d_x$.

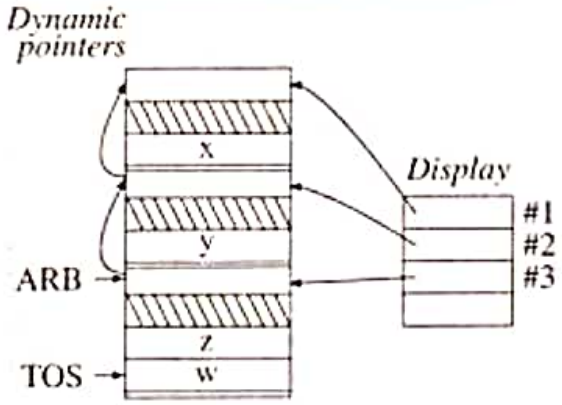


Fig. 6.6 Accessing nonlocal variables using display

Symbol table requirements

In order to implement dynamic allocation and access, a compiler should perform the following tasks while compiling the use of a name v in *b_current*, the block being compiled:

1. Determine the static nesting level of *b_current*.
2. Determine the variable designated by the name v. This should be done in accordance with the scope rules.
3. Determine the static nesting level of the block in which v is defined. Determine the value of d_v.
4. Generate the code indicated in (6.1) or (6.2).

We present a simple scheme to implement functions 1, 2 and 3. This scheme uses the extended stack model of Section 2.2.1 to organize the symbol table. When the start of block *b_current* is encountered during compilation, a new record is pushed on the stack. It contains

1. Nesting level of *b_current*
2. Symbol table for *b_current*.

The reserved pointer of the new record points to the previous record in the stack. This record contains the symbol table of the static ancestor of *b_current*. Each entry in the symbol table contains a variable's name, type, length and displacement in the AR.

Scope rules are implemented by searching the name v referenced in *b_current* in the symbol table as follows: Symbol table in the topmost record of the stack is searched first. Existence of name v there implies that v designates a local variable of *b_current*. If an entry for v does not exist there, the previous record in the stack is searched. It contains the symbol table for the Level 1 ancestor of *b_current*. Existence of v in it implies that v is a variable declared in the Level 1 ancestor block, and not redeclared in *b_current*. If v is not found there, it is searched in the previous

record of the stack, i.e. in the symbol table of the Level 2 ancestor, and so on. When v is found in a symbol table, its displacement d_v in the AR is obtained from its symbol table entry. The nesting level of the block defining v is obtained from the first field of the stack record containing the symbol table. Code can now be generated to implement the access to variable v as in (6.1) or (6.2).

Example 6.12 Figure 6.7 shows the symbol table for the program of Ex. 6.9. For simplicity, each symbol table entry only shows a symbol and its displacement in AR. The search for variable x accessed in the statement `x := z;` terminates on finding the entry of x in the symbol table for block A. The nesting level of A is 1. $d_x = 2$ and nesting level of C, the current block, is 3. This information is sufficient to generate code as in Ex. 6.10 or Ex. 6.11.

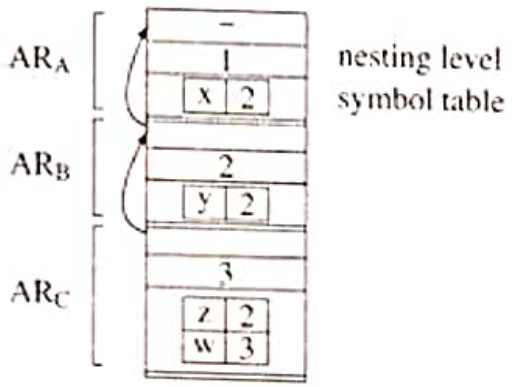


Fig. 6.7 Stack structured symbol table

Recursion

Recursive procedures (or functions) are characterized by the fact that many invocations of a procedure coexist during the execution of a program. A copy of the local variables of the procedure must be allocated for each invocation. This does not pose any problems when a stack model of memory allocation is used because an activation record is created for every invocation of a procedure or function.

Example 6.13 A Pascal program `sample2` contains a function `fib` which computes the nth term of the Fibonacci series.

```
Program sample2 (input, output);
   var
      a,b : integer;
   function fib(n) : integer;
      var
         x : integer;
      begin
         if n > 2 then
            x := fib(n-1) + fib(n-2);
         else x := 1;
         return(x);
```

```
    end fib;
    begin
       fib(4);
end.
```

For the call `fib(4)`, the function makes recursive calls `fib(3)` and `fib(2)`, each of which makes further recursive calls, and so on. Assuming that a formal parameter is allocated memory analogous to a local variable, Fig. 6.8 shows the allocation status after the recursive call `fib(2)`. Creation of multiple AR's for function `fib` is handled naturally by the actions at block entry. Note that the static and dynamic pointers in the activation record for `fib(4)`, i.e. in AR^1_{fib} are identical because `sample2` is both a static and dynamic ancestor of this invocation. However static and dynamic pointers in the activation records for the calls `fib(3)` and `fib(2)`, i.e. in AR^2_{fib} and AR^3_{fib}, are different because their static and dynamic ancestors are different.

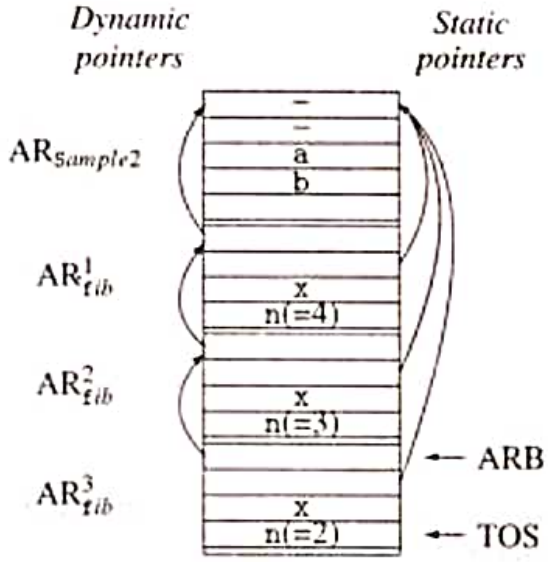


Fig. 6.8 Memory allocation in recursion

Limitations of stack based memory allocation

The stack based memory allocation model is not adequate for program controlled memory allocation, e.g. when the **new** or **dispose** statements of Pascal or `calloc` and `free` functions of C are used to allocate and free memory areas. The compiler must resort to the use of heaps for such allocation. Accesses to variables are now implemented through pointers associated with individual variables rather than with AR's. Free lists or garbage collection are used to implement the reuse of deallocated memory. The stack model is also inadequate for multi-activity programs, i.e. concurrent programs, because in such a program, program blocks may be entered in exited in a non-LIFO manner.

6.2.3 Array Allocation and Access

When an n dimensional array is organized in a computer's memory, the order in which the array elements are arranged in the memory is governed by the rules of the PL. In our discussion, we shall assume array elements to be ordered such that elements occupying adjoining memory locations always differ in the first subscript. For two dimensional arrays, this implies a column-wise arrangement of elements. Figure 6.9 shows the allocation for an array a [5, 10]. It is obvious that static memory allocation is feasible only if dimension bounds of an array are known at compilation time.

```
a [1, 1]
   :
a [5, 1]
a [1, 2]
   :
a [5, 2]
   :
a [1, 10]
   :
a [5, 10]
```

Fig. 6.9 Memory allocation for a 2 dimensional array

The arrangement of array elements in the memory determines the memory mapping to be used in computing the address of an array element. In Fig. 6.9, it has been assumed that the lower bound of each dimension to be 1. Under this assumption, address of an array element a $[s_1, s_2]$ can be determined as follows:

$$\text{A}d.\ \text{a}[s_1, s_2] \quad = \quad \text{A}d.\ \text{a}[1,1] + \{(s_2 - 1) \times n + (s_1 - 1)\} \times k$$

where n is the number of rows in the array and k is the number of memory locations (words or bytes) occupied by each element of the array. For a general two dimensional array

$$\text{a}\ [\ l_1 : u_1,\ l_2 : u_2\ ]$$

where l_i and u_i are the lower and upper bounds of the ith subscript, respectively; address of a $[s_1, s_2]$ is given by the formula

$$\text{A}d.\ \text{a}[s_1, s_2] \quad = \quad \text{A}d.\ \text{a}[l_1, l_2] + \{(s_2 - l_2) \times (u_1 - l_1 + 1) + (s_1 - l_1)\} \times k$$

Defining $range_i$ to be the range of the ith subscript, we have

$$\begin{aligned} range_1 &= u_1 - l_1 + 1, \\ range_2 &= u_2 - l_2 + 1, \text{ and} \end{aligned}$$

$$\begin{aligned} Ad.\ \mathtt{a}[s_1,s_2] &= Ad.\ \mathtt{a}[l_1,l_2] + \{(s_2 - l_2) \times range_1 + (s_1 - l_1)\} \times k \\ &= Ad.\ \mathtt{a}[l_1,l_2] - (l_2 \times range_1 + l_1) \times k \\ &\quad + (s_2 \times range_1 + s_1) \times k \\ &= Ad.\ \mathtt{a}[0,0] + (s_2 \times range_1 + s_1) \times k \end{aligned} \tag{6.3}$$

Note that a [0,0]—which we will call the *base element* of array a—does not exist unless $l_i \leq 0\ \forall i$. However, this fact is irrelevant to the address calculation formula. Generalizing for an m dimensional array

$$\begin{aligned} Ad.\mathtt{a}[s_1,..,s_m] &= Ad.\mathtt{a}[0,..,0] \\ &\quad + \{((\ldots(s_m \times range_{m-1} + s_{m-1}) \times range_{m-2} \\ &\quad + s_{m-2}) \times range_{m-3} + ..) \times range_1 + s_1\} \times k \qquad (6.4) \\ &= Ad.\mathtt{a}[0,..,0] + \Sigma_{i=1}^{m} s_i \times (\Pi_{j=1}^{i-1} range_j) \times k \qquad (6.5) \end{aligned}$$

where $\Pi_{j=1}^{i-1} range_j = 1$ for $i = 1$. Values of l_i, u_i and $range_i$ for all dimentions of the array can be computed at compilation time (if subscript bounds are constants), or at array allocation time. These values can be stored in an array descriptor called a *dope vector* (DV). Figure 6.10 shows the dope vector format for an m dimensional array. If dimension bounds are known at compilation time, the dope vector needs to exist only during program compilation; it is accommodated in the symbol table entry for the array and its contents are used while generating code for an array reference (see Ex. 6.14). In some machine architectures, (6.5) can be computed more efficiently than (6.4) if $\forall j\ \Pi_{j=1}^{i-1} range_j \times k$ is a precomputed constant. In such cases $\Pi_{j=1}^{i-1} range_j \times k$ can be computed and stored in the dope vector instead of $range_j$.

Ad. a [0,...,0]		
No. of dimensions (m)		
l_1	u_1	$range_1$
l_2	u_2	$range_2$
⋮	⋮	⋮
l_m	u_m	$range_m$

Fig. 6.10 Dope vector

Example 6.14 The code generated for an array reference a [i, j] of a two dimensional array a [1:5, 1:10] is shown in Fig. 6.11. Note that the various subscript bounds and subscript ranges used here become constants of the generated program. Provision is made to call an error routine if any subscript value falls outside the corresponding range.

If array dimension bounds are not known during compilation, the dope vector has to exist during program execution. The number of dimensions of an array determines the format and size of its DV. The dope vector is allocated in the AR of a block and

```
MOVER     AREG, I
COMP      AREG, ='5'
BC        GT, ERROR_RTN      Error if i>5
COMP      AREG, ='1'
BC        LT, ERROR_RTN      Error if i<1
MOVER     AREG, J
COMP      AREG, ='10'
BC        GT, ERROR_RTN      Error if j>10
COMP      AREG, ='1'
BC        LT, ERROR_RTN      Error if j<1
MULT      AREG, ='5'         Multiply by range1
ADD       AREG, I
MULT      AREG, ='2'         Multiply by element size
ADD       AREG, ADDR_ABASE   Add Address of a[0,0]
```

Fig. 6.11 Code for an array reference

d_{DV}, its displacement in AR, is noted in the symbol table entry of the array. The array is allocated dynamically by repeating step 6 of Table 6.1 for the array. Its start address, values of l_j, u_j and $range_j$ are entered in the DV. The generated code uses d_{DV} to access the contents of DV to check the validity of subscripts and compute the address of an array element.

Example 6.15 Figure 6.12 shows the memory allocation for the following program segment:

```
var
   x,y : real;
   alpha : array [l_1:u_1, l_2:u_2] of integer;
   i,j : integer;
begin
   alpha[i,j] := ...;
```

where l_1, l_2, u_1 and u_2 are nonlocal variables. Since the dimension bounds are not known during compilation, the dope vector must exist during execution. Hence DV is allocated in the AR. d_{alpha_DV}, its displacement in AR, is thus known at compilation time. Array alpha is allocated when the program segment is activated. The address of its base element is recorded in its DV. The generated code uses knowledge of d_{alpha_DV} to access the contents of DV. For example, 4(ARB) is the *address* field in DV, 6(ARB) gives information about its first dimension, etc. Note that the field *no. of dimensions* may be omitted from DV since this information is implicit in the generated code.

EXERCISE 6.2

1. Develop an algorithm to build the stack structured symbol table shown in Fig. 6.7.
2. In this section we have not described how memory mapping is performed for records of Pascal, Cobol or structures of C. Refer to Dhamdhere (1997) for a discussion of the same.

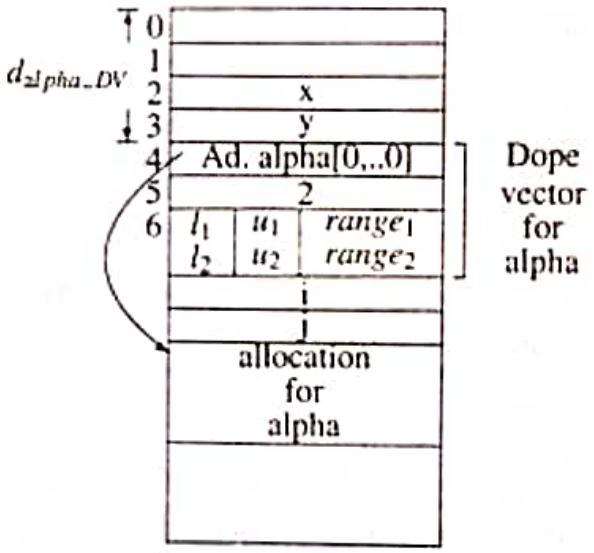


Fig. 6.12 Dope vector for array with dynamic dimension bounds

3. Many languages support bit and character strings with dynamically varying lengths. Would these language features interfere with the use of a stack model for memory allocation?
4. For the recursive procedure `fib`, show contents of the *Display* before and after each recursive call.

6.3 COMPILATION OF EXPRESSIONS

6.3.1 A Toy Code Generator for Expressions

The major issues in code generation for expressions are as follows:

1. Determination of an evaluation order for the operators in an expression.
2. Selection of instructions to be used in the target code.
3. Use of registers and handling of partial results.

The evaluation order of operators depends on operator precedences in an obvious way—an operator which precedes its left and right neighbours must be evaluated before either of them. Hence, a feasible evaluation order is the order in which operators are reduced during a bottom-up parse, or the reverse of the order in which operators are produced during a top down parse. (Other feasible evaluation orders are discussed in the next section.) In the following discussion the use of a bottom up parser to determine the evaluation order has been assumed.

The choice of an instruction to be used in the target code depends on the following:

1. The *type* and *length* of each operand
2. The *addressability* of each operand, i.e. *where* the operand is located and *how* it can be accessed.

The addressability of an operand indicates *where* an operand is located and *how* it can be accessed.

We introduce the notion of an *operand descriptor* to maintain the type, length and addressability information for each operand. Thus the choice of an instruction can be made by analysing an operator and the descriptors of its operands.

A *partial result* is the value of some subexpression computed while evaluating an expression. In the interests of efficiency, partial results are maintained in CPU registers as far as possible. However, some of them have to be moved to memory if the number of results exceeds the number of available CPU registers. An important issue in code generation is when and how to move partial results between memory and CPU registers, and how to know which partial result is contained in a register. We use a *register descriptor* to maintain information for the latter purpose.

We develop a toy code generator for expressions using a YACC like bottom-up parser generator. For simplicity we present a code generator which uses a single CPU register. However, it can be easily extended to handle multiple registers.

Operand descriptors

An operand descriptor has the following fields:

1. *Attributes:* Contains the subfields *type*, *length* and *miscellaneous information*.
2. *Addressability:* Specifies *where* the operand is located, and *how* it can be accessed. It has two subfields
   (a) *Addressability code:* Takes the values 'M' (operand is in memory), and 'R' (operand is in register). Other addressability codes, e.g. address in register ('AR') and address in memory ('AM'), are also possible, however we do not consider them in this section.
   (b) *Address:* Address of a CPU register or memory word.

An operand descriptor is built for every operand participating in an expression, i.e. for *id*'s, constants and partial results. A descriptor is built for an *id* when the *id* is reduced during parsing. A partial result pr_i is the result of evaluating some operator op_j. A descriptor is built for pr_i immediately after code is generated for operator op_j. For simplicity we assume that all operand descriptors are stored in an array called *Operand_descriptor*. This enables us to designate a descriptor by its index in *Operand_descriptor*, e.g. descriptor #*n* is the descriptor in *Operand_descriptor*[*n*].

Example 6.16 The code generated for the expression a*b is as follows:

```
MOVER      AREG, A
MULT       AREG, B
```

Three operand descriptors are used during code generation. Assuming a, b to be integers occupying 1 memory word, these are:

1	(int,1)	M, addr(a)	Descriptor for a
2	(int,1)	M, addr(b)	Descriptor for b
3	(int,1)	R, addr(AREG)	Descriptor for a*b

A skeleton of the code generator is shown in Fig. 6.13. Operand descriptor numbers are used as attributes of NT's. The routine `build_descriptor` called from the semantic action for the production <*F*> ::= <*id*> builds a descriptor for <*id*> and returns its entry number in *Operand_descriptor*. The code generation routine generates code for an operator and builds an operand descriptor for the partial result. Semantic actions of other rules merely copy the operand descriptors as attributes of NT's.

```
%%
E   :  E + T   {$$ = codegen('+', $1, $3)}
    |  T       {$$ = $1}
    ;
T   :  T * F   {$$ = codegen('*', $1, $3)}
    |  F       {$$ = $1}
    ;
F   :  id      {$$ = build_descriptor($1)}
    ;
%%
 build_descriptor(operand)
 {
   i = i + 1;
   operand_descr[i] = ((type), (addressability_code, address))
                                  of operand;
   return i;
 }
```

Fig. 6.13 Skeleton of the code generator

Register descriptors

A register descriptor has two fields

1. *Status:* Contains the code *free* or *occupied* to indicate register status.
2. *Operand descriptor #:* If *status* = *occupied*, this field contains the descriptor # for the operand contained in the register.

Register descriptors are stored in an array called *Register_descriptor*. One register descriptor exists for each CPU register.

Example 6.17 The register descriptor for AREG after generating code for a*b as in Ex. 6.16 would be

Occupied	#3

This indicates that register AREG contains the operand described by descriptor #3.

Generating an instruction

When an operator op_i is reduced by the parser, the function `codegen` is called with op_i and descriptors of its operands as parameters. A single instruction can be generated to evaluate op_i if the descriptors indicate that one operand is in a register and the other is in memory. If both operands are in memory, an instruction is generated to move one of them into a register. This is followed by an instruction to evaluate op_i.

Saving partial results

If all registers are occupied (i.e. they contain partial results) when operator op_i is to be evaluated, a register r is freed by copying its contents into a *temporary location* in the memory. r is now used to evaluate operator op_i. For simplicity we assume that an array `temp` is declared in the target program (i.e. in the generated code) to hold partial results. A partial result is always stored in the next free entry of `temp`. Note that when a partial result is moved to a temporary location, the descriptor of the partial result must change. The *operand descriptor #* field of the register descriptor is used to achieve this.

Example 6.18 Consider the expression `a*b+c*d`. After generating code for `a*b`, the operand and register descriptors would be as shown in Ex. 6.16 and 6.17. After the partial result `a*b` is moved to a temporary location, say `temp[1]`, the operand descriptors must become

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	M, addr(temp[1])

to indicate that the value of the operand described by operand descriptor # 3 (viz. `a*b`) has been moved to memory location `temp[1]`.

Figure 6.14 shows the complete code generation routine. It first checks the addressabilities of its operands to decide whether one of them exists in a register. If so, it generates a single instruction to perform the operation. If none of the operands is in a register, it needs to move one operand into the register before performing the operation. For this purpose, it first frees the register if it is occupied and changes the operand descriptor of the result it contained previously. (Note how it uses the register descriptor for this purpose.) It now moves one operand into the register and generates an instruction to perform the operation. At the end of code generation, it builds a descriptor for the partial result.

Example 6.19 Code generation steps for the expression `a*b+c*d` are shown in Tab. 6.3, where the superscript of an NT shows the operand descriptor # used as its attribute, and the notation $<id>_v$ represents the token $<id>$ constructed for symbol v.

Figure 6.15(a) shows the parse tree and operand and register descriptors after step 8. Figure 6.15(b) shows the parse tree and descriptors after step 9. Note that operand descriptor #3 has been changed to indicate that the partial result `a*b` has been moved

```
codegen(operator, opd1, opd2)
{
   if opd1.addressability_code = 'R'
         /* Code generation -- case 1 */
      if operator = '+' generate 'ADD AREG, opd2';
         /* Analogous code for other operators */
   else if opd2.addressability_code = 'R'
         /* Code generation -- case 2 */
      if operator = '+' generate 'ADD AREG, opd1';
         /* Analogous code for other operators */
   else
         /* Code generation -- case 3 */
      if Register_descr.status = 'Occupied'
         /* Save partial result */
         generate  ('MOVEM AREG, Temp[j]');
         j = j + 1;
         Operand_descr[Register_descr.Operand_descriptor#]
                  = (<type>, (M, Addr(Temp[j])));
      /* Generate code */
      generate 'MOVER AREG, opd1';
      if operator = '+' generate 'ADD AREG, opd2';
         /* Analogous code for other operators */
   /* Common part -- Create a new descriptor
   Saying operand value is in register AREG */
   i = i + 1;
   operand_descr[i] = (<type>,('R', Addr(AREG)));
   Register_descr = ('Occupied', i);
   return i;
}
```

Fig. 6.14 Code generation routine

to the temporary location `temp[1]`. The register descriptor now points to operand descriptor #6 which describes the partial result `c*d`.

To see how temporary locations are used to hold partial results, consider the source string `a*b+c*d*(e+f)+c*d`. Figure 6.16 shows the expression tree for the string wherein operator numbers indicate the bottom up evaluation order. Figure 6.17(a) shows the code generated for this string using the code generation routine of Fig. 6.14. The contents of the temporary locations and their usage are summarized in Table 6.4.

The general rule concerning the use of partial results is that the partial result saved last is always used before the results saved earlier. For example, the value of `c*d` is used before the value of `a*b`. (It is easy to prove the rule by contradiction—if this fact does not hold, we must have evaluated the operators in some order other than the bottom up evaluation order!) It is therefore possible to use a LIFO data structure,

Table 6.3 Code generation actions for `a*b+c*d`

Step no.	*Parsing action*	*Code generation action*
1.	$<id>_a \rightarrow F^1$	Build descriptor # 1
2.	$F^1 \rightarrow T^1$	–
3.	$<id>_b \rightarrow F^2$	Build descriptor # 2
4.	$T^1 * F^2 \rightarrow T^3$	Generate `MOVER AREG, A` `MULT AREG, B` Build descriptor # 3
5.	$T^3 \rightarrow E^3$	–
6.	$<id>_c \rightarrow F^4$	Build descriptor # 4
7.	$F^4 \rightarrow T^4$	–
8.	$<id>_d \rightarrow F^5$	Build descriptor # 5

9.	$T^4 * F^5 \rightarrow T^6$	Generate `MOVEM AREG, TEMP_1`
		`MOVER AREG, C`
		`MULT AREG, D`
		Build descriptor # 6

10.	$E^3 + T^6 \rightarrow E^7$	Generate `ADD AREG, TEMP_1`

Table 6.4

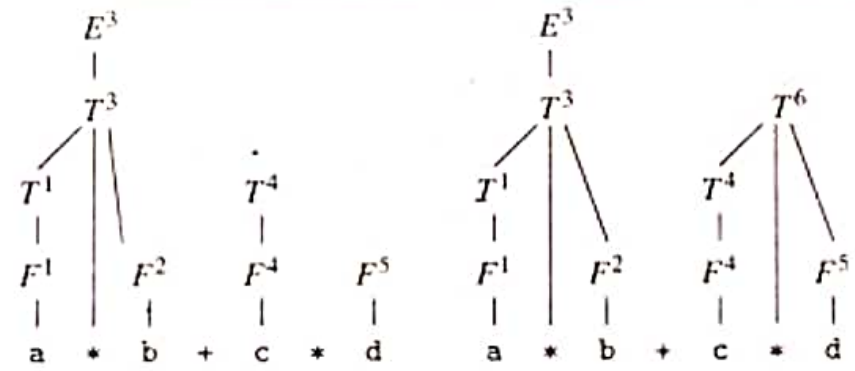
Temporary	*Contents*	*Used in*
temp [1]	Value of node 1 (i.e. value of `a*b`)	evaluating node 5
temp [2]	Value of node 2	evaluating node 4
temp [3]	Value of node 5	evaluating node 7

i.e. a stack, to organize the temporary locations. Use of a stack permits the reuse of temporary locations during the evaluation of an expression, thus reducing the number of temporary locations required. Figure 6.17(b) shows the code for the expression resulting from the reuse of *temp* [1]. The contents of the temporary locations and their usage are summarized in Table 6.5.

Note that *temp* [1] is reused to store the value of node 5. This is realized as follows: When the value contained in *temp* [2] is used in evaluating node 4, *temp* [2] is marked free (see the 9th instruction in Fig. 6.17(b)). Similarly *temp* [1] is marked free

Table 6.5

Temporary	*Contents*	*Used in*
temp [1]	Value of node 1	evaluating node 5
temp [2]	Value of node 2	evaluating node 4
temp [1]	Value of node 5	evaluating node 7



Operand descriptors

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	R, addr(AREG)
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)

(a)

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	M, addr(temp[1])
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)
6	(int,1)	R, addr(AREG)

(b)

Register descriptor

Occ.	3

(a)

Occ.	6

(b)

Fig. 6.15 Code generation actions

when the value contained in it is used in evaluating node 5 (see the 10th instruction). While generating the code for node 6, the partial result contained in the register is saved in the first available location. This happens to be *temp* [1].

To implement a stack of temporaries, variable `i` of routine `codegen` is used as a stack pointer. Now, `i` needs to be decremented whenever an operand of an instruction is a temporary location. If dynamic memory allocation is used, it is best to locate the stack *after* the activation record for a block. Now we can increment/decrement TOS instead of variable `i`. Figure 6.18 illustrates this arrangement.

EXERCISE 6.3.1

1. Extend the toy code generator to incorporate the following:
   (a) Parentheses in an expression.
   (b) Non-commutative operators like '−' and '/'.
2. Extend the toy code generator to handle multiple registers in the CPU. Show various steps in the code generation for the expression `(a+b)/(c+d)` using 2 CPU registers.

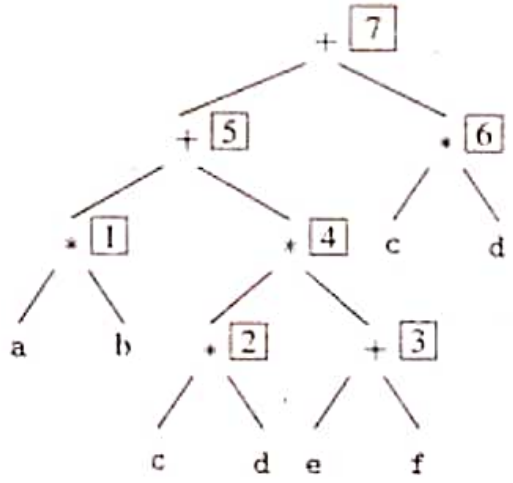


Fig. 6.16 Expression tree

```
MOVER   AREG, A          MOVER   AREG, A
MULT    AREG, B          MULT    AREG, B
MOVEM   AREG, TEMP_1     MOVEM   AREG, TEMP_1
MOVER   AREG, C          MOVER   AREG, C
MULT    AREG, D          MULT    AREG, D
MOVEM   AREG, TEMP_2     MOVEM   AREG, TEMP_2
MOVER   AREG, E          MOVER   AREG, E
ADD     AREG, F          ADD     AREG, F
MULT    AREG, TEMP_2     MULT    AREG, TEMP_2
ADD     AREG, TEMP_1     ADD     AREG, TEMP_1
MOVEM   AREG, TEMP_3     MOVEM   AREG, TEMP_1
MOVER   AREG, C          MOVER   AREG, C
MULT    AREG, D          MULT    AREG, D
ADD     AREG, TEMP_3     ADD     AREG, TEMP_1
```

(a) (b)

Fig. 6.17 Illustration for temporary location usage

6.3.2 Intermediate Codes for Expressions

Postfix strings

In the postfix notation, each operator appears immediately after its last operand. Thus, a binary operator op_i appears after its second operand. If any of its operands is itself an expression involving an operator op_j, op_j must appear before op_i. Thus operators can be evaluated in the order in which they appear in the string.

Example 6.20 Consider the following strings:

$$\text{Source string}: \vdash \mathtt{a} \overset{\boxed{2}}{+} \mathtt{b} \overset{\boxed{1}}{*} \mathtt{c} \overset{\boxed{5}}{+} \mathtt{d} \overset{\boxed{4}}{*} \mathtt{e} \overset{\boxed{3}}{\uparrow} \mathtt{f} \dashv \tag{6.6}$$

Fig. 6.18 Stack of temporaries in AR_P

```
                    [1][2]      [3][4][5]
Postfix string : ├a b c * + d e f ↑ * + ┤
```

The numbers appearing over the operators indicate their evaluation order. The second operand of the operator '+' marked [2] in the source string is the expression b*c. Hence its operands b, c and its operator '*' appear before '+' in the postfix string.

The postfix string is a popular intermediate code in non-optimizing compilers due to ease of generation and use. We perform code generation from the postfix string using a stack of operand descriptors. Operand descriptors are pushed on the stack as operands appear in the string. When an operator with arity k appears in the string, k descriptors are popped off the stack. A descriptor for the partial result generated by the operator is now pushed on the stack. Thus in Ex. 6.20, the operand stack would contain descriptors for a , b and c when '*' is encountered. The stack contains the descriptors for a and the partial result b*c when '+' is encountered.

Conversion from infix to postfix is performed using a slight modification of Algorithm 3.4. Each stack entry simply contains an operator. Operands appearing in the source string are copied into the postfix string straightaway. The TOS operator is popped into the postfix string if TOS operator -> current operator.

Triples and quadruples

A *triple* is a representation of an elementary operation in the form of a pseudo-machine instruction

Operator	*Operand 1*	*Operand 2*

Triples are numbered in some convenient manner. Each operand of a triple is either a variable/constant or the result of some evaluation represented by another triple. In the latter case, the operand field contains that triple's number. Conversion of an infix string into triples can be achieved by a slight modification of Algorithm 3.4.

Example 6.21 Figure 6.19 contains the triples for the expression string (6.6). [1] in the operand field of triple 2 indicates that the operand is the value of b*c represented by triple number 1.

	operator	*operand 1*	*operand 2*
1	*	b	c
2	+	[1]	a
3	↑	e	f
4	*	d	[3]
5	+	[2]	[4]

Fig. 6.19 Triples for string 6.6

A program representation called *indirect triples* is useful in optimizing compilers. In this representation, a table is built to contain all distinct triples in the program. A program statement is represented as a list of triple numbers. This arrangement is useful to detect the occurrences of identical expressions in a program. For efficiency reasons, a hash organization can be used for the table of triples. The indirect triples representation provides memory economy. It also aids in certain forms of optimization, viz. common subexpression elimination.

Example 6.22 Figure 6.20 shows the indirect triples' representation for the program segment

```
z := a+b*c+d*e↑f;
y := x+b*c;
```

Use of `b*c` in both statements is reflected by the fact that triple number 1 appears in the list of triples for both statements. As we shall see in Section 6.5.2, this fact is used to advantage in local common subexpression elimination.

	operator	*operand 1*	*operand 2*
1	*	b	c
2	+	[1]	a
3	↑	e	f
4	*	d	[3]
5	+	[2]	[4]
6	+	x	[1]

triples' table

stmt no.	*triple nos.*
1	1,2,3,4,5
2	1,6

statement table

Fig. 6.20 Indirect triples

A *quadruple* represents an elementary evaluation in the following format:

Operator	*Operand 1*	*Operand 2*	*Result name*

Here, *result name* designates the result of the evaluation. It can be used as the operand of another quadruple. This is more convenient than using a number (as in the case of triples) to designate a subexpression.

	operator	*operand 1*	*operand 2*	*result name*
1	*	b	c	t_1
2	+	t_1	a	t_2
3	↑	e	f	t_3
4	*	d	t_3	t_4
5	+	t_2	t_4	t_5

Fig. 6.21 Quadruples

Example 6.23 Quadruples for the expression string (6.6) are shown in Fig. 6.21.

Note that $t_1, t_2, \ldots t_5$ in Fig. 6.21 are not temporary locations for holding partial results. They are result names. Some of these become temporary locations when common subexpression elimination is implemented. For example, if an expression `x+b*c` were also present in the program, then `b*c` may be a common subexpression. t_1 would then become a compiler generated temporary location. We discuss details of optimization by elimination of common subexpressions in Section 6.5.2.

Expression trees

We have so far assumed that operators are evaluated in the order determined by a bottom up parser. This evaluation order may not lead to the most efficient code for an expression. Hence a compiler back end must analyse an expression to find the best evaluation order for its operators. An *expression tree* is an abstract syntax tree (see Section 3.2) which depicts the structure of an expression. This representation simplifies the analysis of an expression to determine the best evaluation order.

Example 6.24 Figure 6.22 shows two alternative codes to evaluate the expression `(a+b)/(c+d)`. The code in part (b) uses fewer MOVER/MOVEM instructions. It is obtained by deviating from the evaluation order determined by a bottom up parser.

```
MOVER   AREG, A          MOVER   AREG, C
ADD     AREG, B          ADD     AREG, D
MOVEM   AREG, TEMP_1     MOVEM   AREG, TEMP_1
MOVER   AREG, C          MOVER   AREG, A
ADD     AREG, D          ADD     AREG, B
MOVEM   AREG, TEMP_2     DIV     AREG, TEMP_1
MOVER   AREG, TEMP_1
DIV     AREG, TEMP_2
```

(a) (b)

Fig. 6.22 Alternative codes for `(a+b)/(c+d)`

A two step procedure is used to determine the best evaluation order for the operations in an expression. The first step associates a *register requirement label* (RR

label) with each node in the expression. It indicates the number of CPU registers required to evaluate the subtree rooted at the node without moving a partial result to memory. Labelling is performed in a bottom up pass of the expression tree. The second step, which consists of a top down pass, analyses the RR labels of the child nodes of a node to determine the order in which they should be evaluated.

Algorithm 6.1 (Evaluation order for operators)

1. Visit all nodes in an expression tree in post order (i.e., such that a node is visited *after* all its children).

   For each node n_i

   (a) If n_i is a leaf node then

   if n_i is the left operand of its parent then RR(n_i) := 1;

   else RR(n_i) := 0;

   (b) If n_i is not a leaf node then

   If RR($l_child_{n_i}$) $\neq$ RR($r_child_{n_i}$) then

   RR(n_i) := max (RR($r_child_{n_i}$), RR($l_child_{n_i}$));

   else RR(n_i) := RR($l_child_{n_i}$) + 1;

2. Perform the procedure call *evaluation_order* (*root*) (See Fig. 6.23), which prints a postfix form of the source string in which operators appear in the desired evaluation order.

```
procedure evaluation_order (node);
    if node is not a leaf node then
        if RR(l_child_node) ≤ RR(r_child_node) then
            evaluation_order (r_child_node);
            evaluation_order (l_child_node);
        else
            evaluation_order (l_child_node);
            evaluation_order (r_child_node);
    print node;
end evaluation_order;
```

Fig. 6.23 Procedure *evaluation_order*

Let RR=q for the root node. This implies that the evaluation order can evaluate the expression without moving any partial result(s) to memory if q CPU registers are available. It thus provides the most efficient way to evaluate the expression. When the number of available registers $< q$, some partial results have to be saved in memory. However, the evaluation order still leads to the most efficient code. The code generation algorithm is described in Dhamdhere (1997) and Aho, Sethi, Ullman (1986).

Example 6.25 Figure 6.24 shows the expression tree for the string `f+(x+y)*((a+b)/(c-d))`. The boxed numbers indicate operator positions in

the source string. The RR label of each node is shown as the superscript of the operand or operator at that node. The evaluation order according to algorithm 6.1 is [6], [4], [5], [2], [3], [1]. This evaluation order is determined as follows: When *evaluation_order* is called with the root node, i.e. operator [1], as the parameter, it decides that the right child of the node should be evaluated before its left child since RR([3]) > RR(node for f). This leads to a recursive call with node [3] as the parameter. Here RR([5]) > RR([2]) leads to a recursive call with node [5] as the parameter. At node [5] also decision is made to visit its right child before its left child. This leads to the postfix string `cd-ab+/xy+*f+` in which operators appear in the evaluation order mentioned above.

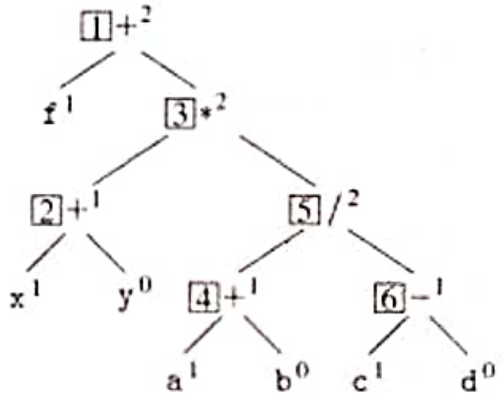


Fig. 6.24 Register requirement labels

EXERCISE 6.3.2

1. Modify the code generator of Fig. 6.13 to
   (a) Generate a postfix string as an IR.
   (b) Generate an expression tree as an IR.

6.4 COMPILATION OF CONTROL STRUCTURES

Definition 6.3 (Control structure) *The* control structure *of a programming language is the collection of language features which govern the sequencing of control through a program.*

The control structure of a PL consists of constructs for control transfer, conditional execution, iteration control and procedure calls. In this section we outline the fundamentals of control structure compilation and describe the compilation of procedure calls in some detail.

Control transfer, conditional execution and iterative constructs

Control transfers implemented through conditional and unconditional **goto**'s are the most primitive control structure. When the target language of a compiler is a machine language, compilation of control transfers is analogous to the assembly of forward or backward **goto**'s in an assembly language program. Hence similar techniques

based around the use of a label table can be used. When the target language is an assembly language (as is mostly the case in practice), a statement **goto lab** can be simply compiled as the assembly statement `BC ANY, LAB`. In the following, the target language of a compiler is assumed to be an assembly language.

Control structures like **if**, **for** or **while** cause significant semantic gap between the PL domain and the execution domain because the control transfers are implicit rather than explicit. This semantic gap is bridged in two steps. In the first step a control structure is mapped into an equivalent program containing explicit **goto**'s. Since the destination of a **goto** may not have a label in the source program, the compiler generates its own labels and puts them against the appropriate statements. Figure 6.25 illustrates programs equivalent to the **if** and **while** statements wherein the labels int_1, int_2 are introduced by the compiler for its own purposes. In the second step, these programs are translated into assembly programs. Note that the first step need not be carried out explicitly. It can be implied in the compilation action, as seen in Ex. 6.26.

```
if (e1) then                    if (ē1) then goto int1;
   S1;                             S1;
else                   ⇒        goto int2;
   S2;                       int1: S2;
S3;                          int2: S3;
```

(a)

```
while (e2) do                int3: if (ē2) then goto int4;
   S1;                             S1;
   S2;                 ⇒           S2;
   ⋮                               ⋮
   Sn;                             Sn;
endwhile;                          goto int3;
<next statement>             int4: <next statement>
```

(b)

Fig. 6.25 Control structure compilation

Example 6.26 Compilation of an **if** statement proceeds as follows (see Fig. 6.25): The compiler selects a pair of labels, viz. int_1 and int_2 in Fig. 6.25, to be associated with the **else** statement and the end of the **if** statement, respectively. Code is generated to evaluate the expression $\bar{e}_1$. This is followed by an instruction `BC ..., int1` which transfers control to int_1 if $\bar{e}_1$ is *true*. Code is now generated for the statements in the **then** part, followed by an instruction `BC ANY, int2`. int_1 is now associated with the first statement of the **else** clause. int_2 is similarly associated with the statement following the **if** statement. The resulting code is:

if (e_1) **then**		{ instructions for $\bar{e}_1$ }
		BC ..., int_1 {Branch if true}
S_1:		{ instructions for S_1 }
	$\Rightarrow$	BC ANY, int_2
else		
S_2:		int_1: { instructions for S_2 }
S_3:		int_2: { instructions for S_3 }

6.4.1 Function and Procedure Calls

A function call, viz. the call on fn_1 in the statement

```
x := fn_1(y,z)+b*c;
```

executes the body of `fn_1`, and returns its value to the calling program. In addition, the function call may also result in some side effects.

Definition 6.4 (Side effect) *A* side effect *of a function (procedure) call is a change in the value of a variable which is not local to the called function (procedure).*

A procedure call only achieves side effects, it does not return a value. Since all considerations excepting the return of a value are analogous for function and procedure calls, in the following only compilation of function calls are discussed.

Example 6.27 A side effect of the call `fn_1(y,z)` could be a change in the value of an actual parameter, viz. y or z, or of a nonlocal variable of `fn_1`.

While implementing a function call, the compiler must ensure the following:

1. Actual parameters are accessible in the called function.
2. The called function is able to produce side effects according to the rules of the PL.
3. Control is transferred to, and is returned from, the called function.
4. The function value is returned to the calling program.
5. All other aspects of execution of the calling programs are unaffected by the function call.

The compiler uses a set of features to implement function calls. These are described below.

1. *Parameter list:* The parameter list contains a *descriptor* for each actual parameter of the function call. The notation D_p is used to represent the descriptor corresponding to formal parameter p.
2. *Save area:* The called function saves the contents of CPU registers in this area before beginning its execution. The register contents are restored from this area before returning from the function.

3. *Calling conventions:* These are execution time assumptions shared by the called function and its caller(s). The conventions include the following:
   (a) How the parameter list is accessed
   (b) How the save area is accessed
   (c) How the transfers of control at call and return are implemented
   (d) How the function value is returned to the calling program.

   Most machine architectures provide special instructions to implement items (c) and (d).

Calling conventions

In a static memory allocation environment, the parameter list and the save area are allocated in the calling program. The calling conventions require the addresses of the function, the parameter list and the save area to be contained in specific CPU registers at the time of call. Let r_{par_list} denote the register containing the address of the parameter list and let $(d_{D_p})_{par_list}$ denote the displacement of D_p in the parameter list. $(d_{D_p})_{par_list}$ is computed while processing the formal parameter list of the function and is stored in the symbol table entry of p. During execution, D_p has the address $< r_{par_list} > + (d_{D_p})_{par_list}$. Hence every reference to p in the function body is compiled using this address. At return, the function value may be returned in a CPU register or in a memory location.

In a dynamic memory allocation environment, the calling program constructs the parameter list and the save area on the stack. These become a part of the called function's AR when its execution is initiated. Start of the parameter list has a known displacement d_{par_list} in the AR. For every formal parameter p, the displacement of D_p in the AR, denoted as $(d_{D_p})_{AR}$, is computed and stored in the symbol table entry of p. During execution, D_p has the address $<ARB> + (d_{D_p})_{AR}$. Ex. 6.28 illustrates calling conventions in the static and dynamic allocation environments.

Example 6.28 Figure 6.26 illustrates the calling conventions used in IBM mainframe systems to implement a call on function `fn_1` with the formal parameters a and b. The use of registers can be summarized as follows:

Register no.	Purpose
0	Function value
1	Address of parameter list
13	Address of save area
14	Return address
15	Address of function

Figure 6.27 illustrates a simplified version of the calling conventions used in a dynamic memory allocation environment. The calling program constructs the parameter list on the stack before invoking the function. At call, ARB is set to point at a location prior to the parameter list. Thus the parameter list is accessible as part of the function's AR. Register contents are now saved in the save area. Before returning, the function

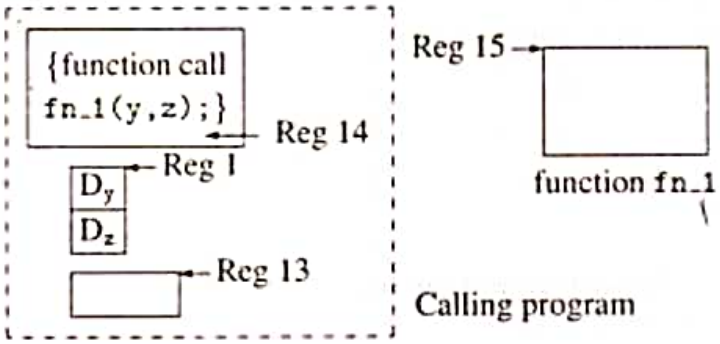


Fig. 6.26 Calling conventions in static memory allocation

value is stored in the AR. At return, TOS is set to <ARB>+2. Thus, the function value is available at the top of the run-time stack. (The stack locations allocated to the static and dynamic pointers would be wasted on return from the function. To avoid this, a practical compiler would put these pointers *after* the function value in the AR format.)

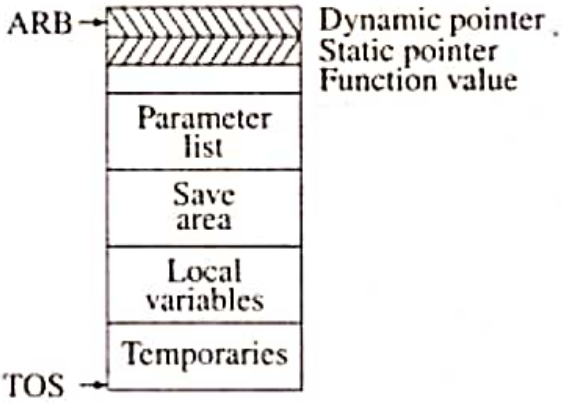


Fig. 6.27 Calling conventions in dynamic memory allocation

Parameter passing mechanisms

Language rules for parameter passing define the semantics of parameter usage inside a function, thereby defining the kind of side effects a function can produce on its actual parameters. The side effect characteristics and the execution efficiency of common parameter passing mechanisms are briefly described.

Call by value

In this mechanism, values of actual parameters are passed to the called function. These values are assigned to the corresponding formal parameters. Note that the passing of values only takes place in one direction—from the calling program to the called function. If a function changes the value of a formal parameter, the change is not reflected on the corresponding actual parameter. Thus a function cannot produce

any side effects on its parameters.

Call by value is commonly used for built-in functions of the language. Its main advantage is its simplicity. A called function may allocate memory to a formal parameter and copy the value of the actual parameter into this location at every call. Thus a formal parameter appears like an initialized local variable, hence the compiler can treat it as if it were a local variable. This simplifies compilation considerably. This mechanism is very efficient if parameters are scalar variables.

Call by value-result

This mechanism extends the capabilities of the call by value mechanism by copying the values of formal parameters back into corresponding actual parameters at return. Thus, side effects are realized *at return*. This mechanism inherits the simplicity of the call by value mechanism but incurs higher overheads.

Call by reference

In this mechanism, the address of an actual parameter is passed to the called function. If the parameter is an expression, its value is computed and stored in a temporary location and the address of the temporary location is passed to the called function. If the parameter is an array element, its address is similarly computed at the time of call.

The parameter list is thus a list of addresses of actual parameters. At every access of a formal parameter in the function, the address of the corresponding actual parameter is obtained from the parameter list. This is used as the address of the formal parameter. The code generated to access the value of formal parameter p is

1. $r \leftarrow$ <ARB> + $(d_{D_p})_{AR}$ or $r \leftarrow < r_{par_list} > + (d_{D_p})_{par_list}$.
2. Access the value using the address contained in register r.

Thus each access to a parameter in the function body incurs the overheads of step 1. Step 2 produces instantaneous side effects if the address in r is used on the left hand side of an assignment. This mechanism is very popular because it has 'cleaner' semantics than call by value-result.

Example 6.29 Consider a function

```
function alpha (a,b : integer) : integer;
   z := a;
   i := i+1;
   b := a+5;
return;
end alpha;
```

where z, i are nonlocal variables of alpha. Let the parameters be passed by reference and let alpha be called as alpha(d[i],x). The address of d[i] is evaluated at the time of call and put into the parameter list. This address is used as the address of a in alpha. Note that a change in the value of i in the body of alpha does not

change this address. The assignment `b := ..` changes the value of `x` instantaneously. This is important in case a nested function call accesses `x` as a nonlocal variable. (If parameters are passed by value-result, the side effect on `x` would be realized at return. Thus a nested function call would access the old value of `x` rather than its new value assigned in function `alpha`.)

Call by name

This parameter transmission mechanism has the same effect as if every occurrence of a formal parameter in the body of the called function is replaced by the name of the corresponding actual parameter.

Example 6.30 In the program of Ex. 6.29, name substitution implies replacement of `a` by `d[i]` in the body of `alpha` during the execution of `alpha`. Thus the call `alpha (d[i],x)` has the same effect as execution of the statements

```
z := d[i];
i := i+1;
x := d[i]+5;
```

Apart from achieving instantaneous side effects, call by name has the implication of changing its actual parameter corresponding to a formal parameter *during* the execution of a function, e.g. formal parameter `a` in the above code corresponds to two different elements of `d` at different times in the execution of `alpha`.

Call by name is implemented as follows: Each parameter descriptor in the parameter list is the address of a routine which computes the address of the corresponding actual parameter. The code generated for every access of formal parameter `p` in the function body is

1. $r \leftarrow <\text{ARB}> + (d_{D_p})_{AR}$ or $r \leftarrow < r_{par_list} > + (d_{D_p})_{par_list}$.
2. Call the function whose address is contained in r.
3. Use the address returned by the function to access `p`.

As seen in Ex. 6.30, the actual parameter corresponding to a formal parameter can change dynamically during the execution of a function. This makes the call by name mechanism immensely powerful. However, the high overheads of step 2 in the code make it less attractive in practice.

Most languages use call by value for built-in functions. C also uses it for programmer defined functions. Pascal permits a programmer to choose between call by value and call by reference. Fortran and PL/I use call by reference only. Ada leaves the choice between call by value-result and call by reference to the compiler writer. Algol-60 used call by value and call by name.

EXERCISE 6.4

1. Many block structured languages permit nonlocal `goto`'s. Comment on the compilation of such **goto**'s. (Hint: Think of memory deallocation.)

2. Most PLs forbid control transfers into the iteration control constructs, e.g. `goto`'s to statements situated in **for** loops. Devise a scheme to detect such control structure violations.
3. Compare and contrast the following parameter passing mechanisms in terms of execution efficiency and power to produce side effects.
   (a) call by value-result
   (b) call by reference
   (c) call by name.
4. What are the results produced by the following program for the different parameter passing mechanisms discussed in this section?

```
a : array[1..20] of real;      procedure sub_1(x,y,z,w);
k := 7;                        x, y : integer;
i := 11;                       y := x;
a[11] := 0.0;                  x := x + 1;
a[12] := -0.5;                 z := w;
sub_1(i,k,a[i],i+k);           return;
print k,i,a[i],a[k];           end;
```

5. A compiler uses a stack to compile nested control structures in a program. Each entry in the stack indicates information about one control structure. This information is used for error detection and code generation. An entry for a control structure is pushed/popped when the start/end of the control structure is encountered during compilation.

   Design a scheme to compile **if** and **for** statements using this approach. (Hint: see Ex. 6.26.)
6. Study the **case** control structure of Pascal and its generated code. Comment on how the code differs from nested **if** statements. Design a code generation model for the **case** statement.

6.5 CODE OPTIMIZATION

Code optimization aims at improving the execution efficiency of a program. This is achieved in two ways:

1. Redundancies in a program are eliminated.
2. Computations in a program are rearranged or rewritten to make it execute efficiently.

It is axiomatic that code optimization must not change the meaning of a program. Two points concerning the scope of optimization should also be noted. First, optimization seeks to improve a program rather than the algorithm used in a program. Thus replacement of an algorithm by a more efficient algorithm is beyond the scope of optimization. Second, efficient code generation for a specific target machine (e.g. by fully exploiting its instruction set) is also beyond its scope; it belongs in the back end of a compiler. The optimization techniques are thus independent of both the PL and the target machine.

Figure 6.28 contains a schematic of an optimizing compiler. It differs from the schematic in Fig. 1.12 in the presence of the optimization phase. The front end generates an IR which could consist of triples, quadruples or ASTs. The optimization phase transforms this to achieve optimization. The transformed IR is input to the back end.

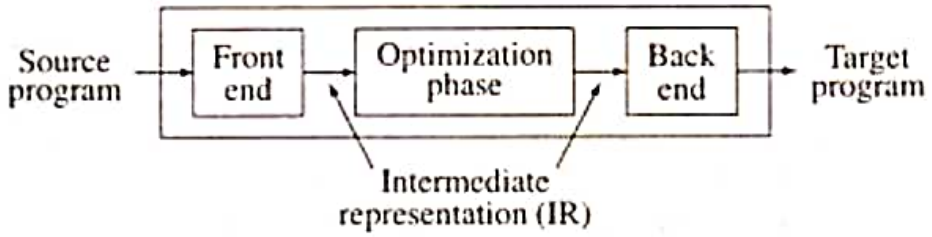


Fig. 6.28 Schematic of an optimising compiler

The structure of a program and the manner in which it manipulates its data provide vital clues for optimization. The compiler analyses a program to collect information concerning these aspects. The cost and benefits of optimization depend on how exhaustively a program is analysed. Experience with the Fortran 'H' optimizing compiler for the IBM/360 system (Lowry, Medlock, 1969) gives a quantitative feel for the cost and benefits of code optimization. The compiler was found to consume 40 percent extra compilation time due to optimization. The optimized program occupied 25 percent less storage and executed three times as fast as the unoptimized program.

6.5.1 Optimizing Transformations

An *optimizing transformation* is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning. Optimizing transformations are classified into *local* and *global* transformations depending on whether they are applied over small segments of a program consisting of a few source statements, or over larger segments consisting of loops or function bodies. The reason for this distinction is the difference in the costs and benefits of the optimizing transformations. A few optimizing transformations commonly used in compilers are discussed below.

Compile time evaluation

Execution efficiency can be improved by performing certain actions specified in a program during compilation itself. This eliminates the need to perform them during execution of the program, thereby reducing the execution time of the program. *Constant folding* is the main optimization of this kind. When all operands in an operation are constants, the operation can be performed at compilation time. The result of the operation, also a constant, can replace the original evaluation in the program. Thus, an assignment a := 3.14157/2 can be replaced by a := 1.570785, thereby eliminating

a division operation. An instance of compile time evaluation can be found in array address arithmetic. The products $(\Pi_{j=1}^{i-1} range_j) \times k$ can be evaluated at compilation time (see Section 6.2.3) if $range_j$ $\forall j$ and k are constants. This avoids evaluation of these products while executing every array reference.

Elimination of common subexpressions

Common subexpressions are occurrences of expressions yielding the same value. (Such expressions are called *equivalent expressions*.) Let CS_i designate a set of common subexpressions. It is possible to eliminate an occurrence $e_j \in CS_i$ if, no matter how the evaluation of e_j is reached during the execution of the program, the value of some $e_k \in CS_i$ would have been already computed. Provision is made to save this value and use it at the place of occurrence of e_j.

Example 6.31

```
                          t := b*c;
a := b*c                  a := t;
- - -              ⇒      - - -
x := b*c+5.2;             x := t+5.2;
```

Here CS_i contains the two occurrences of `b*c`. The second occurrence of `b*c` can be eliminated because the first occurrence of `b*c` is always evaluated before the second occurrence is reached during execution of the program. The value computed at the first occurrence is saved in `t`. This value is used in the assignment to `x`.

This optimization is implemented as follows: First, expressions which yield the same value are identified. Many compilers simplify this task by restricting its scope to congruent, i.e. identical, subexpressions. These can be easily identified using triples or quadruples (see indirect triples, Section 6.3.2). Their equivalence is determined by considering whether their operands have the same values in all occurrences. Occurrences of the subexpression which satisfy the criterion mentioned earlier for expressions e_j can be eliminated. Some compilers also use rules of algebraic equivalence in common subexpression elimination. In the following program:

```
..:= b*c ..
d := b;
..:= d*c..
```

`d*c` is a common subexpression since `d` has the same value as `b`. Use of algebraic equivalence improves the effectiveness of optimization. However, it also increases the cost of optimization.

Dead code elimination

Code which can be omitted from a program without affecting its results is called *dead code*. Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.

Example 6.32 An assignment `x := <exp>` constitutes dead code if the value assigned to `x` is not used in the program, no matter how control flows after executing this assignment. Note that <*exp*> constitutes dead code only if its execution does not produce side effects, i.e. only if it does not contain function or procedure calls.

Frequency reduction

Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times. For example, the transformation of *loop optimization* moves loop invariant code out of a loop and places it prior to loop entry.

Example 6.33

```
                                    x := 25*a;
for i := 1 to 100 do                for i := 1 to 100 do
begin                               begin
   z := i;              ⇒             z := i;
   x := 25*a;                         y := x+z;
   y := x+z;                        end;
end;
```

Here `x := 25*a;` is loop invariant. Hence in the optimized program it is computed only once before entering the **for** loop. `y := x+z;` is not loop invariant. Hence it cannot be subjected to frequency reduction.

Strength reduction

The strength reduction optimization replaces the occurrence of a time consuming operation (a 'high strength' operation) by an occurrence of a faster operation (a 'low strength' operation), e.g. replacement of a multiplication by an addition.

Example 6.34 In Fig. 6.29, the 'high strength' operator '`*`' in `i*5` occurring inside the loop is replaced by a low strength operator '`+`' in `itemp+5`.

```
                                    itemp := 5;
for i := 1 to 10 do                 for i := 1 to 10 do
begin                               begin
   - - -                               - - -
   k := i*5;                           k := itemp;
   - - -                  ⇒            - - -
                                       itemp := itemp+5;
end;                                end;
```

Fig. 6.29 Strength reduction

Strength reduction is very important for array accesses occurring within program loops. For example, an array reference `a[i,j]` within a Pascal loop gives rise to the

high strength computation `i*n` in the address arithmetic, where n is the number of rows in the array (see Section 6.2.3). Strength of `i*n` can be reduced as in Fig. 6.29. Note that strength reduction optimization is not performed on operations involving floating point operands because finite precision of floating point arithmetic cannot guarantee equivalence of results after strength reduction.

Local and global optimization

Optimization of a program is structured into the following two phases:

1. *Local optimization:* The optimizing transformations are applied over small segments of a program consisting of a few statements,
2. *Global optimization:* The optimizing transformations are applied over a program unit, i.e. over a function or a procedure.

Local optimization is a preparatory phase for global optimization. It can be performed by the front end while converting a source program into the IR (see Fig. 6.36). Local optimization also simplifies certain aspects of global optimization. For example, let a program segment seg_i contain n occurrences of an expression `a+b`. After local common subexpression elimination has been performed over seg_i, global optimization only needs to consider elimination of the first occurrence of `a+b`—other occurrences of `a+b` are either not redundant, or would have been already eliminated!

6.5.2 Local Optimization

Local optimization provides limited benefits at a low cost. The scope of local optimization is a *basic block* which is an 'essentially sequential' segment in the source program. The cost of local optimization is low because the sequential nature of the basic block simplifies the analysis needed for optimization. The benefits are limited because certain optimizations, e.g. loop optimization, are beyond the scope of local optimization.

Definition 6.5 (Basic block) *A* basic block *is a sequence of program statements* $(s_1, s_2, \ldots, s_n)$ *such that only* s_n *can be a transfer of control statement and only* s_1 *can be the destination of a transfer of control statement.*

A basic block b is a program segment with a single entry point. If control reaches statement s_1 during program execution, all statements $s_1, s_2, \ldots, s_n$ will be executed. The 'essentially sequential' nature of a basic block simplifies optimization. Example 6.35 discusses this aspect. We shall also see this aspect while discussing the value numbering technique to perform local optimization.

Example 6.35 Consider the following program segment:

```
                                        t := x*y;
          a := x*y;                     a := t;
          - - -          ⇒              - - -
          b := x*y;                     b := t;
lab_i :   c := x*y;          lab_i :    c := x*y;
```

where lab_i is a label. Local optimization identifies two basic blocks in the program. The first block extends up to the statement `b := x*y;`. Its optimization leads to elimination of the second occurrence of `x*y`. The third occurrence is not eliminated because it belongs to a different basic block. If the label lab_i did not exist, the entire program segment would constitute a single basic block and the third occurrence `x*y` can also be eliminated during local optimization.

Value numbers

Value numbers provide a simple means to determine if two occurrences of an expression in a basic block are equivalent. The value numbering technique is applied on the fly while identifying basic blocks in a source program. A value number vn_{alpha} is associated with variable `alpha`. It identifies the last assignment to `alpha` processed so far. Thus, the value number of variable `alpha` changes on processing an assignment `alpha := ...`. Now two expressions e_i and e_j are equivalent if they are congruent and their operands have the same value numbers.

For simplicity all statements of a basic block are numbered in some convenient manner. If statement n, the current statement being processed, is an assignment to `alpha`, we set vn_{alpha} to n. A new field is added to each symbol table entry to hold the value number of a variable. The IC for a basic block is a list of quadruples stored in a tabular form. Each operand field in a quadruple holds the pair (*operand, value number*). A boolean flag *save* is associated with each quadruple to indicate whether its value should be saved for use elsewhere in the program. The flag is initialized to *false* in every new quadruple entered in the table.

Let expression e be a subexpression of e', the expression being compiled. While forming a quadruple for e, the value numbers of its operands are copied from the symbol table. The new quadruple is now compared with all existing quadruples in the IC. Existence of a matching quadruple q_i in the IC implies that the current occurrence of expression e has the same value as a previous occurrence represented by quadruple q_i. If a match is found, the newly generated quadruple is not entered in the IC. Instead, the result name of q_i is used as an operand of e'. In effect, this occurrence of e is eliminated from the program. The result name of q_i should now become a compiler generated temporary variable. This requirement is noted by setting the *save* flag of q_i to *true*. During code generation, this flag is checked to see if the value of q_i needs to be saved in a temporary location.

Example 6.36 Figure 6.30 shows the symbol table and the quadruples table during local optimization of the following program:

Symbol table

Symbol	...	*Value number*
y		0
x		15
g		14
z		0
d		5
w		0

Quadruples table

	Operator	*Operand 1*		*Operand 2*		*Result name*	*Use flag*
		Operand	*Value no.*	*Operand*	*Value no.*		
20	:=	g	–	25.2	–	t_{20}	*f*
21	+	z	0	2	–	t_{21}	*f*
22	:=	x	0	t_{21}	–	t_{22}	*f*
23	*	x	15	y	0	t_{23}	*ft*
24	+	t_{23}	–	d	5	t_{24}	*f*
	⋮						
57	:=	w	0	t_{23}	–	t_{57}	*f*

Fig. 6.30 Local optimization using value numbering

stmt no.	*statement*
14	`g := 25.2;`
15	`x := z+2;`
16	`h := x*y+d;`
..	...
34	`w := x*y;`

Local optimization proceeds as follows: All variables are assumed to have the value numbers '0' to start with. Processing of Statements 14 and 15 leads to generation of quadruples numbered 20–22 shown in the table. Value numbers of g, x and y at this stage are 14, 15 and 0, respectively (see the symbol table). Quadruple for `x*y` is generated next, and the value numbers of x and y are copied from the symbol table. This quadruple is assigned the result name t_{23}, and its *save* flag is set to *false*. This quadruple is entered in entry number 23 of the quadruple table. When the statement `w := x*y` (i.e., Statement 34) is processed, the quadruple for `x*y` is formed using the value numbers found in the symbol table entries of x and y. Since these are still 15 and 0, the new quadruple is identical with quadruple 23 in the table. Hence it is not entered in the table. Instead, the *save* flag of quadruple 23 is set to *true*. The only quadruple generated for this statement is therefore the assignment to w (quadruple number 57). While generating code for quadruple 23, its save flag indicates that the

value of expression x*y needs to be saved in a temporary location for later use. t_{23} can itself become the name of this temporary location.

This schematic can be easily extended to implement *constant propagation*, which is the substitution of a variable *var* occurring in a statement by a constant *const*, and constant folding. When an assignment of the form *var* := *const* is encountered, we enter *const* into a table of constants, say in entry *n*, and associate the value number '$-n$' with *var*. Constant propagation and folding is implemented while generating a quadruple if each operand is either a constant or has a negative value number.

Example 6.37 In the following program, variable a is given a negative value number, say '-10', on processing the first statement.

```
a := 27.3;          a := 27.3;
- - -          ⇒    - - -
b := a*3.0;         b := 81.9;
```

This leads to the possibility of constant propagation and folding in the third statement. The value of a, viz. '27.3', is obtained from the 10th entry of constants table. Its multiplication with 3.0 yields 81.9. This value is treated as a new constant and a quadruple for b := 81.9 is now generated.

6.5.3 Global Optimization

Compared to local optimization, global optimization requires more analysis effort to establish the feasibility of an optimization. Consider global common subexpression elimination. If some expression x*y occurs in a set of basic blocks SB of program P, its occurrence in a block $b_j \in$ SB can be eliminated if the following two conditions are satisfied for every execution of P:

1. Basic block b_j is executed only after some block $b_k \in$ SB has been executed one or more times.
2. No assignments to x or y have been executed after the last (or only) evaluation of x*y in block b_k. (6.7)

Condition 1 ensures that x*y is evaluated before execution reaches block b_j, while condition 2 ensures that the evaluated value is equivalent to the value of x*y in block b_j. The optimization is realized by saving the value of x*y in a temporary location in all blocks b_k which satisfy condition 1.

To ensure that *every possible execution* of program P satisfies conditions 1 and 2 of (6.7), the program is analysed using the techniques of *control flow analysis* and *data flow analysis*. Note the emphasis on the words 'every possible execution'. This requirement is introduced to ensure that the meaning of the program is unaffected by the optimization. In this section we use the word 'always' to imply 'in every possible evaluation'.

6.5.3.1 Program Representation

A program is represented in the form of a *program flow graph*.

Definition 6.6 (Program flow graph (PFG))

A program flow graph *for a program P is a directed graph* $G_P = (N, E, n_0)$ *where*

N	:	*set of basic blocks in P*
E	:	*set of directed edges* (b_i, b_j) *indicating the possibility of control flow from the last statement of* b_i *(the* source node*) to the first statement of* b_j *(the* destination node*)*
n_0	:	*start node of P.*

A basic block, which is a sequence of statements $s_1, s_2, \ldots s_n$, is visualized as a sequence of *program points* $p_1, p_2, \ldots p_n$ such that a statement s_i is said to exist at program point p_i. This notion is used to differentiate between different occurrences of identical statements. For example, an occurrence of e at program point p_i is distinct from the occurrence of e at program point p_j.

6.5.3.2 Control and Data Flow Analysis

The techniques of control and data flow analysis are together used to determine whether the Conditions governing an optimizing transformation are satisfied in a program, e.g. conditions 1 and 2 of (6.7).

Control flow analysis

Control flow analysis analyses a program to collect information concerning its structure, e.g. presence and nesting of loops in the program. Information concerning program structure is used to answer specific questions of interest, e.g. condition 1 of (6.7). The control flow concepts of interest are:

1. *Predecessors and successors:* If $(b_i, b_j) \in E$, b_i is a *predecessor* of b_j and b_j is a *successor* of b_i.

2. *Paths:* A path is a sequence of edges such that the destination node of one edge is the source node of the following edge.

3. *Ancestors and descendants:* If a path exists from b_i to b_j, b_i is an ancestor of b_j and b_j is a descendant of b_i.

4. *Dominators and post-dominators:* Block b_i is a dominator of block b_j if every path from n_0 to b_j passes through b_i. b_i is a post-dominator of b_j if every path from b_j to an exit node passes through b_i.

Table 6.6 Data flow concepts

Data flow concept	*Optimization in which used*
Available expression	Common subexpression elimination
Live variable	Dead code elimination
Reaching definition	Constant and variable propagation

Control flow concetps can be used to answer certain questions in a straightforward manner, e.g. the question posed by condition 1 of (6.7). However, this incurs the overheads of control flow analysis for every expression in the program. It may sometimes be possible to reduce the overheads by restricting the scope of optimization. For example, only those expressions may be considered which occur in a block b_j and a dominator block b_k of b_j. Now condition 1 of (6.7) is automatically satisfied.

Data flow analysis

Data flow analysis techniques analyse the use of data in a program to collect information for the purpose of optimization. This information, called *data flow information*, is computed at the entry and exit of each basic block in G_P. It is used to decide whether an optimizing transformation (see Section 6.5.1) can be applied to a segment of code in the program.

Design of the global optimization phase begins with the identification of an appropriate *data flow concept* to support the application of each optimizing transformation. The data flow information concerning a program entity—for example a variable or an expression—is now a boolean value indicating whether the data flow concept is applicable to that entity. Table 6.6 contains a summary of important data flow concepts used in optimization. The first two data flow concepts are discussed in detail.

Available expressions

The transformation of global common subexpression elimination can be defined as follows: Consider a subexpression `x*y` occurring at program point p_i in basic block b_i. This occurrence can be eliminated if

1. Conditions 1 and 2 of (6.7) are satisfied at entry to b_i.
2. No assignments to `x` or `y` precede the occurrence of `x*y` in b_i.

The data flow concept of *available expressions* is used to implement common subexpression elimination. An expression e is *available* at program point p_i if a value equivalent to its value is always computed before program execution reaches p_i. Thus, the concept of available expressions captures the essence of Conditions 1 and 2 of (6.7). The availability of an expression at entry or exit of basic block b_i is computed using the following rules:

1. Expression e is available at the exit of b_i if

   (i) b_i contains an evaluation of e which is not followed by assignments to any operands of e, or

   (ii) the value of e is available at the entry to b_i and b_i does not contain assignments to any operands of e. (6.8)

2. Expression e is available at entry to b_i if it is available at the exit of each predecessor of b_i in G_P.

Available expressions is termed a *forward* data flow concept because availability at the exit of a node determines availability at the entry of its successor(s). It is an *all paths* concept because availability at entry of a basic block requires availability at the exit of all predecessors.

We use the boolean variables $Avail_in_i$ and $Avail_out_i$ to represent the availability of expression e at entry and exit of basic block b_i, respectively. Further, we associate the following boolean properties with block b_i to summarize the effect of computations situated in it:

$Eval_i$	:	'true' only if expression e is evaluated in b_i and none of its operands are modified following the evaluation
$Modify_i$	:	'true' only if some operand of e is modified in b_i.

$Eval_i$ and $Modify_i$ are determined solely by the computations situated in b_i. Hence they are called local properties of block b_i. $Avail_in_i$ and $Avail_out_i$ are 'global' properties which are computed using the following equations (see Eq. (6.8)):

$$Avail_in_i = \Pi_{b_j \in pred(b_i)} Avail_out_j \quad (6.9)$$

$$Avail_out_i = Eval_i + Avail_in_i \,.\, \neg Modify_i \quad (6.10)$$

where $\Pi_{b_j \in pred(b_i)}$ is the boolean 'and' operation over all predecessors of b_i. This operation ensures that $Avail_in_i$ is true only if $Avail_out$ is true for all predecessors of b_i. Equations (6.9) and (6.10) are called *data flow equations*.

It is to be noted that every basic block in G_P has a pair of equations analogous to (6.9)–(6.10). Thus, we need to solve a system of simultaneous equations to obtain the values of $Avail_in$ and $Avail_out$ for all basic blocks in G_P. Data flow analysis is the process of solving these equations. Iterative data flow analysis is a simple method which assigns some initial values to $Avail_in$ and $Avail_out$, and iteratively recomputes them for all blocks according to Eqs. (6.9)–(6.10) until they converge onto consistent values.

The initial values are:

$$Avail_in_i = \begin{cases} true & \text{if } b_i \in N - \{n_o\} \\ false & \text{if } b_i = n_o \end{cases}$$

$$Avail_out_i = true \quad \forall b_i$$

After solving the system of equations (6.9)–(6.10) for all blocks, an evaluation of expression e can be eliminated from a block b_i if

1. $Avail_in_i = true$, and
2. The evaluation of e in b_i is not preceded by an assignment to any of its operands.

Example 6.38 Available expression analysis for the PFG of Fig. 6.31 gives the following results:

a*b : *Avail_in* = *true* for blocks 2,5,6,7,8,9
Avail_out = *true* for blocks 1,2,5,6,7,8,9,10

x+y : *Avail_in* = *true* for blocks 6,7,8,9
Avail_out = *true* for blocks 5,6,7,8,9

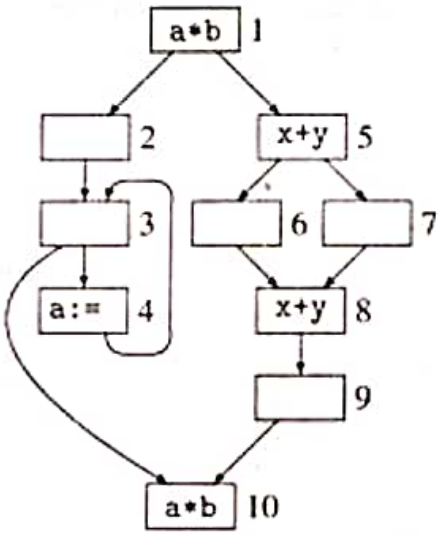


Fig. 6.31 A PFG

The values of *Avail_in* and *Avail_out* for a*b can be explained as follows: The assignment a := .. in block 4 makes $Avail_out_4$ = *false* (eq. (6.10)). This makes $Avail_in_3$ = *false* (eq. (6.9)), which makes $Avail_out_3 = Avail_in_{10}$ = *false*. For x*y, $Avail_in_1$ = *false* makes $Avail_out_1$ = *false*, which makes *Avail_in* = *Avail_out* = *false* for blocks 2, 3, 4 and 10.

Live variables

A variable *var* is said to be *live* at a program point p_i in basic block b_i if the value contained in it at p_i is likely to be used during subsequent execution of the program. If *var* is not live at the program point which contains a definition *var* := ..., the value assigned to *var* by this definition is redundant in the program. Such a definition constitutes dead code which can be eliminated from the program without changing its meaning.

The liveness property of a variable can be determined as follows:

1. Variable v is live at the entry of b_i if

   (i) b_i contains a use of e which is not preceded by assignment(s) to v, or
   (ii) v is live at the exit of b_i and b_i does not contain assignment(s) to v.

2. v is live at exit of b_i if it is live at the entry of some successor of b_i in G_P.

Data flow information concerning live variables can be collected as follows:

$Live_in_i$	:	*var* is live at entry of b_i
$Live_out_i$	:	*var* is live at exit of b_i
Ref_i	:	*var* is referenced in b_i and no assignment of *var* precedes the reference
Def_i	:	An assignment to *var* exists in b_i

$$Live_in_i = Ref_i + Live_out_i \,.\, \neg Def_i \tag{6.11}$$

$$Live_out_i = \Sigma_{b_j \in succ(b_i)} Live_in_j \tag{6.12}$$

where $\Sigma_{b_j \in succ(b_i)}$ is the boolean 'or' operation over all successors of b_i.

Live variables is termed a *backward* data flow concept because availability at the entry of a block determines availability at the exit of its predecessor(s). It is an *any path* concept because liveness at the entry of one successor is sufficient to ensure liveness at the exit of a block. The data flow problem can be solved by iterative data flow analysis using the initializations $Live_in_i = Live_out_i = false\ \forall b_i$.

Example 6.39 In the PFG of Fig. 6.31, variable b is live at the entry of all blocks, a is live at the entry of all blocks excepting block 4 and variables x, y are live at the entry of blocks 1,5,6,7 and 8.

EXERCISE 6.5

1. Write a program to generate code from a table of quadruples (see Fig. 6.30).
2. Build a program flow graph for the following program:

```
z := 5;
w := z;
for i := 1 to 100 do
   x := a*b;
   y := c+d;
   if y < 0 then
      a := 25;
      f := c+d;
   else
      g := w;
      h := a*b+f;
      d := z+10;
```

```
end;
g := c+d;
print g,h,d,x,y;
```

Apply the following transformations to optimize the program:

(a) common subexpression elimination
(b) dead code elimination
(c) constant propagation
(d) frequency reduction.

3. Apply the data flow concepts of available expressions and live variables to the flow graph of the program in problem 2 and verify whether you obtain the same answers as in parts (a) and (b) of problem 2.
4. A definition def_i is an assignment of the form *var* := <*exp*>. def_i located at program point p_i is said to reach a program point p_j if the value assigned to *var* by def_i may be the value of *var* when program execution reaches p_j.

   Write the data flow equations to collect the information concerning reaching definitions. (*Hint:* Consider whether the data flow concept is a forward or backward concept and any path or all paths concept.)
5. Constant propagation is the substitution of a variable *var* by a constant $const_i$ in a statement. State the conditions under which constant propagation can be performed. (*Hint:* See problem 4.)
6. Loop invariant movement of an assignment *var* := <*exp*> (see Ex. 6.33) is performed if the following conditions are met:

   (a) <*exp*> is loop invariant.
   (b) The assignment is the only assignment to *var* inside the loop.
   (c) The assignment dominates all loop exist.
   (d) *var* is not live on entry to the loop.

   Discuss the need for each condition.

6.6 INTERPRETERS

In Section 1.2.2 we have seen that use of interpretation avoids the overheads of compilation. This is an advantage during program development, because a program may be modified very often—in fact, it may be modified between every two executions. However, interpretation is expensive in terms of CPU time, because each statement is subjected to the interpretation cycle. Hence, conventional wisdom warns against interpreting a program with large execution requirements. However, to make an informed decision in practice, we need a quantitative basis for a comparison of compilers and interpreters. We introduce the following notation for this purpose:

t_c : average compilation time per statement
t_e : average execution time per statement
t_i : average interpretation time per statement

Note that both compilers and interpreters analyse a source statement to determine its meaning. During compilation, analysis of a statement is followed by code generation, while during interpretation it is followed by actions which implement its meaning. Hence we could assume $t_c \cong t_i$. t_e, which is the execution time of the compiler generated code for a statement, can be several times smaller than t_c. Let us assume $t_c = 20.t_e$.

Consider a program P. Let $size_P$ and $stmts_executed_P$ represent the number of statements in P and the number of statements executed in some execution of P, respectively. We use these parameters to compute the CPU time required to execute a program using compilation or interpretation. Example 6.40 illustrates how this can be done.

Example 6.40 Let $size_P = 200$. For a specific set of data, let program P execute as follows: 20 statements are executed for initialization purposes. This is followed by 10 iterations of a loop containing 8 statements, followed by the execution of 20 statements for printing the results. Thus, $stmts_executed_P = 20 + 10 \times 8 + 20 = 120$. Thus,

Total execution time using the compilation model

$$= 200.t_c + 120.t_e$$
$$\cong 206.t_c.$$

Total execution time using the interpretation model

$$= 120.t_i$$
$$\cong 120.t_c.$$

Clearly, interpretation is beneficial in this case.

Use of interpreters

Use of interpreters is motivated by two reasons—efficiency in certain environments and simplicity. The findings of Ex. 6.40 concerning efficiency can be generalized as follows: It is better to use interpretation for a program P if P is modified between executions, and $stmts_executed_P < size_P$. These conditions are satisfied during program development, hence interpretation should be preferred during program development. In all other situations, it is best to use compilation.

It is simpler to develop an interpreter than to develop a compiler because interpretation does not involve code generation (see Section 6.6.1). This simplicity makes interpretation more attractive in situations where programs or commands are not executed repeatedly. Hence interpretation is a popular choice for commands to an operating system or an editor. User interfaces of many software packages prefer interpretation for similar reasons.

6.6.1 Overview of Interpretation

In this section we discuss an interpretation schematic which implements the meaning of a statement without generating code for it. The interpreter consists of three main components:

1. *Symbol table:* The symbol table holds information concerning entities in the source program.
2. *Data store:* The data store contains values of the data items declared in the program being interpreted. The data store consists of a set of components $\{comp_i\}$. A component $comp_i$ is an array named $name_i$ containing elements of a distinct type $type_i$.
3. *Data manipulation routines:* A set of data manipulation routines exist. This set contains a routine for every legal data manipulation action in the source language.

On analysing a declaration statement, say a statement declaring an array `alpha` of type `typ`, the interpreter locates a component $comp_j$ of its data store, such that $type_j$ = `typ`. `alpha` is now mapped into a part of $name_j$ (this is 'memory allocation'). The memory mapping for `alpha` is remembered in its symbol table entry. An executable statement is analysed to identify the actions which constitute its meaning. For each action, the interpreter finds the appropriate data manipulation routine and invokes it with appropriate parameters. For example, the meaning of statement `a := b+c;` where `a`, `b`, `c` are of the same type can be implemented by executing the calls

```
add (b, c, result);
assign (a, result);
```

in the interpreter.

This schematic has two important advantages. First, the meaning of a source statement is implemented through execution of the interpreter routines rather than through code generation. This simplifies the interpreter. Second, avoiding generation of machine language instructions helps to make the interpreter portable. For example, if the interpreter is itself coded in a higher level programming language it can be ported to a new computer system with ease.

6.6.2 A Toy Interpreter

This section discusses the design and operation of an interpreter for Basic written in Pascal. The interpreter is itself a Pascal program (we will call this the *interpreter program*), which is compiled by a Pascal compiler. The data store of the interpreter consists of two large arrays named `rvar` and `ivar` which are used to store real and integer values respectively. Last few locations in the `rvar` and `ivar` arrays are used as stacks for expression evaluation with the help of the pointers `r_tos` and `i_tos` respectively. (Note that these stacks grow 'upwards' in the arrays.)

The interpreter program contains a set of data manipulation routines for use in expression evaluation (see Fig. 6.32)—one routine for each kind of legal subexpression in the language. A routine performs all subtasks involved in the evaluation of a subexpression, namely type compatibility checks, type conversion and evaluation of the subexpressions. Since the interpreter is a Pascal program, it can be ported to any computer system possessing a Pascal compiler.

```
program interpreter (source,output);
   type
      symentry = record
         symbol : array [1..10] of character;
         type : character;
      address : integer
      end;

   var
      symtab : array [1..100] of symentry;
      rvar : array [1..100] of real;
      ivar : array [1..100] of integer;
      r_tos : 1..100;
      i_tos : 1..100;

   procedure assignint (addr1 : integer; value : integer);
      begin
         ivar[addr1] := value;
      end;

   procedure add (sym1, sym2 : symentry);
      begin
      ...
      if (sym1.type = 'real' and sym2.type = 'int') then
         addrealint(sym1.address, sym2.address);
      ...
      end;

   procedure addrealint (addr1, addr2 : integer);
      begin
         rvar[r_tos] := rvar[addr1] + ivar[addr2]
      end;

   begin { Main Program }
      r_tos := 100;
      i_tos := 100;
      {Analyse a statement and call
       appropriate procedure}
   end.
```

Fig. 6.32 Basic interpreter written in Pascal

Example 6.41 Consider the basic program

```
real a, b
integer c
let c = 7
let b = 1.2
a = b+c
```

Figure 6.33 illustrates the symbol table for the program and the memory allocation for variables a, b and c. Each symbol table entry contains information about the type and memory allocation for a variable. The values 'real' and '8' in the symbol table entry of a indicate that a is allocated the word `rvar [8]`.

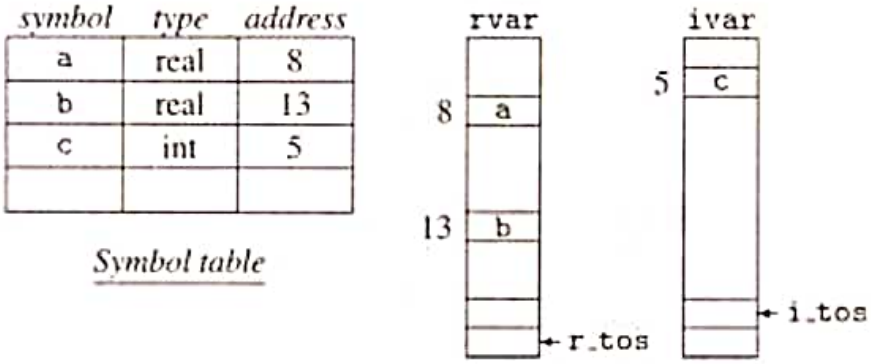


Fig. 6.33 Interpreter data structures

The operation of the interpreter can be explained as follows: The interpreter analyses a source statement to determine its meaning. It implements the meaning by invoking appropriate routines. Consider the assignment statement `c = 7` in the Basic program, where both c and '7' are of type integer. The Pascal procedure `assignint` of Fig. 6.32 is invoked with two parameters. The first parameter is the address of c in `ivar` and the second parameter is the RHS value. Execution of procedure `assignint` effectively executes the Pascal statement

```
ivar [5] := 7;
```

Similarly, the assignment to b is executed as `rvar [13] := 1.2;` by a routine `assignreal`.

Interpretation of `a = b+c` proceeds as follows: The interpreter procedure `add` is called with the symbol table entries of b and c. `add` analyses the types of b and c and decides that procedure `addrealint` will have to be called to realize the addition. `addrealint` now executes a single Pascal statement which is equivalent to

```
rvar [r_tos] := rvar [13] + ivar [5];
```

Note that `b+c` involves a type conversion operation, which is performed implicitly by the above Pascal statement. In other words, while compiling the expression

```
rvar [addr1] + ivar [addr2]
```

in the interpreter program, the Pascal compiler would have made provision for type conversion of the second operand from integer to real. The interpreter simply arranges to execute this statement under the right conditions.

6.6.3 Pure and Impure Interpreters

The schematic of Fig. 6.34(a) is called a *pure* interpreter. The source program is retained in the source form all through its interpretation. This arrangement incurs substantial analysis overheads while interpreting a statement.

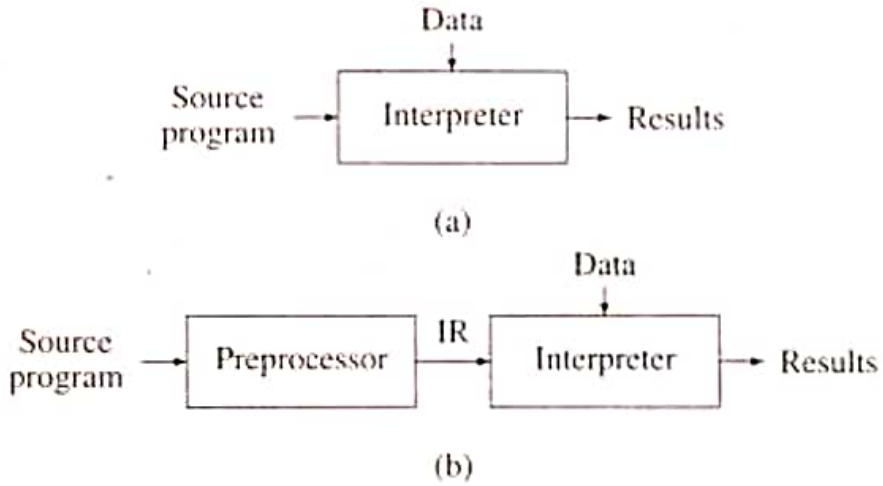


Fig. 6.34 Pure and impure interpreter

An *impure interpreter* performs some preliminary processing of the source program to reduce the analysis overheads during interpretation. Figure 6.34(b) contains a schematic of impure interpretation. The preprocessor converts the program to an intermediate representation (IR) which is used during interpretation. This speeds up interpretation as the code component of the IR, i.e. the IC, can be analysed more efficiently than the source form of the program. However, use of IR also implies that the entire program has to be preprocessed after any modification. This involves fixed overheads at the start of interpretation.

Example 6.42 Postfix notation is a popular intermediate code for interpreters. The intermediate code for a source string `a+b*c` could look like the following:

S #17	S #4	S #29	*	+

where each IC unit resembles a token (see Section 1.3.1.1).

IC of Ex. 6.42 eliminates most of the analysis during interpretation excepting type analysis to determine the need for type conversion. Even this can be eliminated if the preprocessor performs type analysis before generating IC.

Example 6.43 The preprocessor of an interpreter performs type analysis to generate following IC for the expression `a+b*c`, where `a`, `b` are of type real and `c` is of type integer:

S #17	S #4	S #29	$t_{i \to r}$	$*_r$	$+_r$

where the unary operator $t_{i \to r}$ indicates type conversion of an operand (in this case, c) from 'integer' to 'real'. The arithmetic operators are also type specific now.Thus '$*_r$' indicates multiplication in the 'real' representation. This eliminates most analysis during interpretation.

EXERCISE 6.6

1. Modify the interpreter of Ex. 6.41 to use the IC of Ex. 6.43.
2. An interpreter is to be written for a block structured language. Comment on the symbol table management and memory allocation techniques which can be used by the interpreter.
3. Develop an interpretation scheme for nested control structures, e.g. nested `for` statements.
4. In SNOBOL4 programming language, the type of a variable changes dynamically during program execution according to the following rule: The type of a variable x at any point during the execution of a program is the type of the last value assigned to it.

   Comment on the compilation and interpretation of this feature of SNOBOL4.

BIBLIOGRAPHY

Elson (1973), Tennent (1981), Pratt (1983) and Maclennan (1983) have discussed different aspects of the design of programming languages and their influence on programming ease, compilation strategies and execution efficiencies. It is recommended that the reader should acquaint himself with the material covered in one of these texts before reading specialized material.

Gries (1971), Weingarten (1973) and Bauer and Eickel (1974) are important early books on compilers. Later books include those by Lewis, Rosenkrantz and Stearns (1976), Barrett and Couch (1977), Dhamdhere (1997), Tremblay and Sorenson (1984), Aho, Sethi and Ullman (1986), Fischer and LeBlanc (1988), and Watson (1989).

Good discussions on storage allocation can be found in books by Gries (1971) and Dhamdhere (1997). Expression compilation is discussed by Hopgood (1967), Rohl (1975), Dhamdhere (1997) and Aho, Sethi and Ullman (1986). Compilation of control structures and code optimization is discussed by Dhamdhere (1997) and Aho, Sethi and Ullman (1986).

(a) Programming languages

1. Elson, M. (1973): *Concepts of Programming Languages*, Science Research Associates, Chicago.
2. Maclennan, B.J. (1983): *Principles of Programming Languages*, Holt, Rinehart & Winston, New York.
3. Pratt, T.W. (1983): *Programming Languages – Design and Implementation*, Prentice-Hall, Englewood Cliffs.
4. Tennent, R.D. (1981): *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs.

(b) Computer architecture and compilers

1. Doran, R.W. (1979): *Computer Architecture – A Structured Approach*, Academic Press, London.
2. Organick, E.I. (1973): *Computer System Architecture*, Academic Press, London.
3. Wulf, W.A. (1981): "Compilers and computer architecture," *Computer*, **14** (7), 41-48.

(c) Compilers, general

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.
2. Barrett, W.A. and J.D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.
3. Bauer, F.L. and J. Eickel (1974): *Compiler Construction : An Advanced Course*, Springer-Verlag, Berlin.
4. Calingaert, P. (1979): *Assemblers, Compilers and Program Translation*, Computer Science Press, Maryland.
5. Dhamdhere, D.M. (1997): *Compiler Construction – Principles and Practice, 2nd edition*, Macmillan India, New Delhi.
6. Fischer, C.N. and R.J. LeBlanc (1988) : *Crafting a compiler*, Benjamin/ Cummings, Menlo Park, California.
7. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.
8. Hansen, P.B. (1985): *Brinch Hansen on Pascal compilers*, Prentice-Hall, Englewood Cliffs, N.J.
9. Hill, U. (1974): "Special run time organization techniques for Algol-68," in *Compiler Construction : An advanced course*, Bauer, F.L., Eickel, J. (eds), Springer-Verlag, Berlin.
10. Lewis, P.M., D.J. Rosenkrantz, and R.E. Stearns (1976): *Compiler Design Theory*, Addison-Wesley, Reading.
11. Peck, J.E.L. (ed.) (1971): *Algol-68 Implementation*, North Holland, Amsterdam.
12. Tremblay, J.P. and P.G. Sorenson (1984): *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.
13. Watson, D. (1989): *High Level Languages and their Compilers*, Addison-Wesley, Reading.
14. Weingarten, F.W. (1973): *Translation of Computer Languages*, Holden-Day, San Francisco.

(d) Interpreters and related topics

1. Ayres, R.B. and R.L. Derrenbacher (1971): "Partial recompilation," *Proceedings of AFIPS SJCC*, **38**, 497-502.
2. Berthaud, M. and M. Griffiths (1973): "Incremental compilation and conversational interpretation," *Annual Review in Automatic Programming*, **7** (2), 95-114.
3. Brown, P.J. (1979): *Writing Interactive Compilers and Interpreters*, Wiley, New York.
4. Early, J. and P. Caizergues (1972): "A method of incrementally compiling languages with nested statement structure," *Commn. of ACM*, **15** (12), 1040-1044.

5. Klint, P. (1981): "Interpretation techniques," *Software – Practice and Experience*, **11**, 963-973.
6. Kornerup, P., B.B. Kristensen and O.L. Madsen (1980): "Interpretation and code generation based on intermediate languages," *Software – Practice and Experience*, **10** (8), 636-658.
7. McIntyre, T.C. (1978): *Software Interpreters for Micro computers*, Wiley, New York.
8. Ryan, J.L., R.L. Crandall and M.C. Medwedeff (1966): "A conversational system for incremental compilation and execution in a time sharing environment," *Proceedings of AFIPS FJCC*, **29**, 1-22.

CHAPTER 7

Linkers

Execution of a program written in a language L involves the following steps:

1. *Translation* of the program
2. *Linking* of the program with other programs needed for its execution
3. *Relocation* of the program to execute from the specific memory area allocated to it
4. *Loading* of the program in the memory for the purpose of execution.

These steps are performed by different language processors. Step 1 is performed by the translator for language L. Steps 2 and 3 are performed by a *linker* while Step 4 is performed by a *loader*. The terms linking, relocation and loading are defined in a later section.

Figure 7.1 contains a schematic showing steps 1–4 in the execution of a program. The translator outputs a program form called *object module* for the program. The linker processes a set of object modules to produce a ready-to-execute program form, which we will call a *binary program*. The loader loads this program into the memory for the purpose of execution. As shown in the schematic, the object module(s) and ready-to-execute program forms can be stored in the form of files for repeated use.

Translated, linked and load time addresses

While compiling a program P, a translator is given an origin specification for P. This is called the *translated origin* of P. (In an assembly program, the programmer can specify the origin in a `START` or `ORIGIN` statement.) The translator uses the value of the translated origin to perform memory allocation for the symbols declared in P. This results in the assignment of a *translation time address* t_{symb} to each symbol *symb* in the program. The *execution start address* or simply the *start address* of a program is the address of the instruction from which its execution must begin. The start address specified by the translator is the *translated start address* of the program.

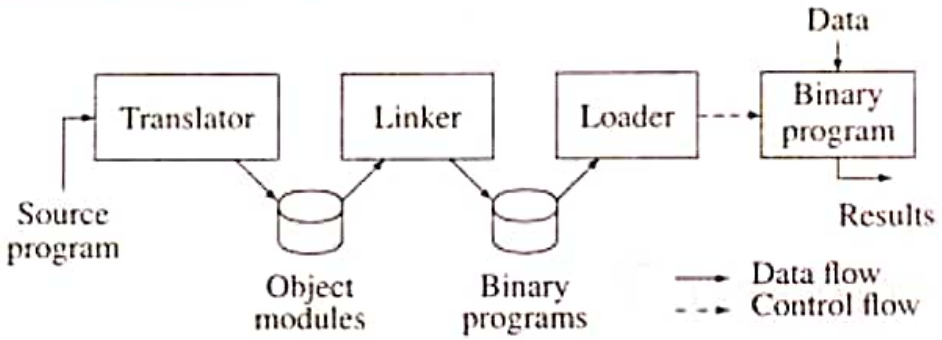


Fig. 7.1 A schematic of program execution

The origin of a program may have to be changed by the linker or loader for one of two reasons. First, the same set of translated addresses may have been used in different object modules constituting a program, e.g. object modules of library rotines often have the same translated origin. Memory allocation to such programs would conflict unless their origins are changed. Second, an operating system may require that a program should execute from a specific area of memory. This may require a change in its origin. The change of origin leads to changes in the execution start address and in the addresses assigned to symbols. The following terminology is used to refer to the address of a program entity at different times:

1. *Translation time* (or *translated*) *address:* Address assigned by the translator.
2. *Linked address:* Address assigned by the linker.
3. *Load time* (or *load*) *address:* Address assigned by the loader.

The same prefixes *translation time* (or *translated*), *linked* and *load time* (or *load*) are used with the origin and execution start address of a program. Thus,

1. *Translated origin:* Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.
2. *Linked origin:* Address of the origin assigned by the linker while producing a binary program.
3. *Load origin:* Address of the origin assigned by the loader while loading the program for execution.

The linked and load origins may differ from the translated origin of a program due to one of the reasons mentioned earlier.

Example 7.1 Consider the assembly program and its generated code shown in Fig. 7.2. The translated origin of the program is 500. The translation time address of LOOP is therefore 501. If the program is loaded for execution in the memory area starting with the address 900, the load time origin is 900. The load time address of LOOP would be 901.

```
        Statement                        Address      Code
        START   500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A                        500)    + 09 0 540
LOOP                                     501)
         ⋮
        MOVER   AREG, ALPHA              518)    + 04 1 000
        BC      ANY, MAX                 519)    + 06 6 000
         ⋮
        BC      LT, LOOP                 538)    + 06 1 501
        STOP                             539)    + 00 0 000
A       DS      1                        540)
TOTAL   DS      1                        541)
        END
```

Fig. 7.2 A sample assembly program and its generated code

7.1 RELOCATION AND LINKING CONCEPTS

7.1.1 Program Relocation

Let AA be the set of absolute addresses—instruction or data addresses—used in the instructions of a program P. AA $\neq \phi$ implies that program P assumes its instructions and data to occupy memory words with specific addresses. Such a program—called an *address sensitive program*—contains one or more of the following:

1. An *address sensitive instruction:* an instruction which uses an address $a_i \in$ AA.
2. An *address constant:* a data word which contains an address $a_i \in$ AA.

In the following, we discuss relocation of programs containing address sensitive instructions. Address constants are handled analogously.

An address sensitive program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin. To execute correctly from any other memory area, the address used in each address sensitive instruction of P must be 'corrected'.

Definition 7.1 (Program relocation) Program relocation *is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.*

If linked origin $\neq$ translated origin, relocation must be performed by the linker. If load origin $\neq$ linked origin, relocation must be performed by the loader. In general, a linker always performs relocation, whereas some loaders do not. For simplicity, in the first part of the chapter it has been assumed that loaders do not perform

relocation—that is, load origin = linked origin. Such loaders are called *absolute loaders*. Hence the terms 'load origin' and 'linked origin' are used interchangeably. However, it would have been more precise to use the term 'linked origin'. (Loaders that perform relocation, i.e. *relocating loaders*, are discussed in Section 7.6.)

Example 7.2 The translated origin of the program in Fig. 7.2 is 500. The translation time address of symbol `A` is 540. The instruction corresponding to the statement `READ A` (existing in translated memory word 500) uses the address 540, hence it is an address sensitive instruction. If the linked origin is 900, `A` would have the link time address 940. Hence the address in the `READ` instruction should be corrected to 940. Similarly the instruction in translated memory word 538 contains 501, the address of `LOOP`. This should be corrected to 901. (Note that the operand addresses in the instructions with the addresses 518 and 519 also need to be corrected. This is explained in Section 7.1.2.)

Performing relocation

Let the translated and linked origins of program P be t_origin_P and l_origin_P, respectively. Consider a symbol *symb* in P. Let its translation time address be t_{symb} and link time address be l_{symb}. The relocation factor of P is defined as

$$relocation_factor_P = l_origin_P - t_origin_P \tag{7.1}$$

Note that $relocation_factor_P$ can be positive, negative or zero.

Consider a statement which uses *symb* as an operand. The translator puts the address t_{symb} in the instruction generated for it. Now,

$$t_{symb} = t_origin_P + d_{symb}$$

where d_{symb} is the offset of *symb* in P. Hence

$$l_{symb} = l_origin_P + d_{symb}$$

Using (7.1),

$$\begin{aligned} l_{symb} &= t_origin_P + relocation_factor_P + d_{symb} \\ &= \underbrace{t_origin_P + d_{symb}} + relocation_factor_P \\ &= t_{symb} + relocation_factor_P \end{aligned} \tag{7.2}$$

Let IRR_P designate the set of instructions requiring relocation in program P. Following (7.2), relocation of program P can be performed by computing the relocation factor for P and adding it to the translation time address(es) in every instruction $i \in IRR_P$.

Example 7.3 For the program of Fig. 7.2

$$\begin{aligned} \text{relocation factor} &= 900 - 500 \\ &= 400. \end{aligned}$$

Relocation is performed as follows: IRR_P contains the instructions with translated addresses 500 and 538. The instruction with translated address 500 contains the address 540 in the operand field. This address is changed to (540+400) = 940. Similarly, 400 is added to the operand address in the instruction with translated address 538. This achieves the relocation explained in Ex. 7.2.

7.1.2 Linking

Consider an application program AP consisting of a set of program units SP = $\{P_i\}$. A program unit P_i interacts with another program unit P_j by using addresses of P_j's instructions and data in its own instructions. To realize such interactions, P_j and P_i must contain public definitions and external references as defined in the following:

Public definition	a symbol *pub_symb* defined in a program unit which may be referenced in other program units
External reference	a reference to a symbol *ext_symb* which is not defined in the program unit containing the reference.

The handling of public definitions and external references is described in the following.

EXTRN and ENTRY statements

The `ENTRY` statement lists the public definitions of a program unit, i.e. it lists those symbols defined in the program unit which may be referenced in other program units. The `EXTRN` statement lists the symbols to which external references are made in the program unit.

Example 7.4 In the assembly program of Fig. 7.2, the `ENTRY` statement indicates that a public definition of `TOTAL` exists in the program. Note that `LOOP` and `A` are not public definitions even though they are defined in the program. The `EXTRN` statement indicates that the program contains external references to `MAX` and `ALPHA`. The assembler does not know the address of an external symbol. Hence it puts zeroes in the address fields of the instructions corresponding to the statements `MOVER AREG, ALPHA` and `BC ANY, MAX`. If the `EXTRN` statement did not exist, the assembler would have flagged references to `MAX` and `ALPHA` as errors.

Resolving external references

Before the application program AP can be executed, it is necessary that for each P_i in SP, every external reference in P_i should be bound to the correct link time address.

Definition 7.2 (Linking) Linking *is the process of binding an external reference to the correct link time address.*

An external reference is said to be *unresolved* until linking is performed for it. It is said to be *resolved* when its linking is completed.

```
        Statement                Address      Code
          START   200
          ENTRY   ALPHA
          - -
          - -
ALPHA     DS      25               231)     + 00 0 025
          END
```

Fig. 7.3 Program unit Q

Example 7.5 Let the program unit of Fig. 7.2 (referred to as program unit P) be linked with the program unit Q described in Fig. 7.3.

Program unit P contains an external reference to symbol ALPHA which is a public definition in Q with the translation time address 231. Let the link origin of P be 900 and its size be 42 words. The link origin of Q is therefore 942, and the link time address of ALPHA is 973. Linking is performed by putting the link time address of ALPHA in the instruction of P using ALPHA, i.e. by putting the address 973 in the instruction with the translation time address 518 in P.

Binary programs

Definition 7.3 (Binary program) *A* binary program *is a machine language program comprising a set of program units SP such that* $\forall P_i \in SP$

1. P_i *has been relocated to the memory area starting at its link origin, and*
2. *Linking has been performed for each external reference in* P_i.

To form a binary program from a set of object modules, the programmer invokes the linker using the command

```
linker   <link origin>, <object module names>
         [, <execution start address> ]
```

where *<link origin>* specifies the memory address to be given to the first word of the binary program. *<execution start address>* is usually a pair (program unit name, offset in program unit). The linker converts this into the linked start address. This is stored along with the binary program for use when the program is to be executed. If specification of *<execution start address>* is omitted the execution start address is assumed to be the same as the linked origin.

Note that a linker converts the object modules in the set of program units SP into a binary program. Since we have assumed link address = load address, the loader simply loads the binary program into the appropriate area of memory for the purpose of execution.

7.1.3 Object Module

The object module of a program contains all information necessary to relocate and link the program with other programs. The object module of a program P consists of 4 components:

1. *Header:* The header contains *translated origin*, *size* and *execution start address* of P.
2. *Program:* This component contains the machine language program corresponding to P.
3. *Relocation table:* (RELOCTAB) This table describes IRR_P. Each RELOCTAB entry contains a single field:

   Translated address : Translated address of an address sensitive instruction.

4. *Linking table* (LINKTAB): This table contains information concerning the public definitions and external references in P.

   Each LINKTAB entry contains three fields:

   Symbol : Symbolic name
   Type : PD/EXT indicating whether public definition or external reference
   Translated address : For a public definition, this is the address of the first memory word allocated to the symbol. For an external reference, it is the address of the memory word which is required to contain the address of the symbol.

Example 7.6 Consider the assembly program of Fig. 7.2. The object module of the program contains the following information:

1. *translated origin* = 500, *size* = 42, *execution start address* = 500.
2. Machine language instructions shown in Fig. 7.2.
3. Relocation table

500
538

4. Linking table

ALPHA	EXT	518
MAX	EXT	519
A	PD	540

Note that the symbol LOOP does not appear in the linking table. This is because it is not declared as a public definition, i.e. it does not appear in an ENTRY statement.

7.2 DESIGN OF A LINKER

7.2.1 Relocation and Linking Requirements in Segmented Addressing

The relocation requirements of a program are influenced by the addressing structure of the computer system on which it is to execute. Use of the segmented addressing structure reduces the relocation requirements of a program.

Example 7.7 Consider the program of Fig. 7.4 written in the assembly language of Intel 8088. The ASSUME statement declares the segment registers CS and DS to be available for memory addressing. Hence all memory addressing is performed by using suitable displacements from their contents. Translation time address of A is 0196. In statement 16, a reference to A is assembled as a displacement of 196 from the contents of the CS register. This avoids the use of an absolute address, hence the instruction is not address sensitive. Now no relocation is needed if segment SAMPLE is to be loaded in the memory starting at the address 2000 because the CS register would be loaded with the address 2000 by a calling program (or by the OS). The effective operand address would be calculated as <CS> + 0196, which is the correct address 2196. A similar situation exists with the reference to B in statement 17. The reference to B is assembled as a displacement of 0002 from the contents of the DS register. Since the DS register would be loaded with the execution time address of DATA_HERE, the reference to B would be automatically relocated to the correct address.

```
Sr. no.                    Statement                    Offset
0001   DATA_HERE   SEGMENT
0002   ABC         DW        25                         0000
0003   B           DW ?                                 0002
  ⋮                  ⋮
0012   SAMPLE      SEGMENT
0013               ASSUME    CS:SAMPLE,
                             DS:DATA_HERE
0014               MOV       AX, DATA_HERE              0000
0015               MOV       DS, AX                     0003
0016               JMP       A                          0005
0017               MOV       AL,B                       0008
  ⋮                  ⋮
0027   A           MOV       AX,BX                      0196
  ⋮                  ⋮
0043   SAMPLE      ENDS
0044               END
```

Fig. 7.4 An 8088 assembly program for linking

Though use of segment registers reduces the relocation requirements, it does not completely eliminate the need for relocation. Consider statement 14 of Fig. 7.4, viz.

```
MOV        AX, DATA_HERE
```

which loads the segment base of DATA_HERE into the AX register preparatory to its transfer into the DS register. Since the assembler knows DATA_HERE to be a segment, it makes provision to load the higher order 16 bits of the address of DATA_HERE into the AX register. However, it does not know the link time address of DATA_HERE, hence it assembles the MOV instruction in the immediate operand format and puts zeroes in the operand field. It also makes an entry for this instruction in RELOCTAB so that the linker would put the appropriate address in the operand field. Inter-segment calls and jumps are handled in a similar way.

Relocation is somewhat more involved in the case of intra-segment jumps assembled in the FAR format. For example, consider the following program:

```
FAR_LAB   EQU     THIS FAR   ; FAR_LAB is a FAR label
          - - -
          JMP     FAR_LAB    ; A FAR jump
```

Here the displacement and the segment base of FAR_LAB are to be put in the JMP instruction itself. The assembler puts the displacement of FAR_LAB in the first two operand bytes of the instruction, and makes a RELOCTAB entry for the third and fourth operand bytes which are to hold the segment base address. A statement like

```
ADDR_A    DW          OFFSET A
```

(which is an 'address constant') does not need any relocation since the assembler can itself put the required offset in the bytes. In summary, the only RELOCTAB entries that must exist for a program using segmented memory addressing are for the bytes that contain a segment base address.

For linking, however, both segment base address and offset of the external symbol must be computed by the linker. Hence there is no reduction in the linking requirements.

7.2.2 Relocation Algorithm

Algorithm 7.1 (Program relocation)

1. *program_linked_origin* := <*link origin*> from `linker` command;
2. For each object module
   (a) *t_origin* := *translated origin* of the object module;
   OM_size := *size* of the object module;
   (b) *relocation_factor* := *program_linked_origin* − *t_origin*;
   (c) Read the machine language program in *work_area*.
   (d) Read RELOCTAB of the object module.
   (e) For each entry in RELOCTAB
      (i) *translated_addr* := address in the RELOCTAB entry;
      (ii) *address_in_work_area* := address of *work_area*
      + *translated_address* − *t_origin*;

(iii) Add *relocation_factor* to the operand address in the word with the address *address_in_work_area*.

(f) *program_linked_origin* := *program_linked_origin* + *OM_size*;

The computations performed in the algorithm are along the lines described in Section 7.1.1. The only new action is the computation of the work area address of the word requiring relocation (step 2(e)(ii)). Step 2(f) increments *program_linked_origin* so that the next object module would be granted the next available load address.

Example 7.8 Let the address of *work_area* be 300. While relocating the object module of Ex. 7.6, *relocation factor* = 400. For the first RELOCTAB entry, *address_in_work_area* = 300 + 500 − 500 = 300. This word contains the instruction for `READ A`. It is relocated by adding 400 to the operand address in it. For the second RELOCTAB entry, *address_in_work_area* = 300 + 538 − 500 = 338. The instruction in this word is similarly relocated by adding 400 to the operand address in it.

7.2.3 Linking Requirements

Features of a programming language influence the linking requirements of programs. In Fortran all program units are translated separately. Hence all subprogram calls and `common` variable references require linking. Pascal procedures are typically nested inside the main program. Hence procedure references do not require linking—they can be handled through relocation. References to built in functions, however, require linking. In C, program files are translated separately. Thus, only function calls that cross file boundaries and references to global data require linking.

A reference to an external symbol `alpha` can be resolved only if `alpha` is declared as a public definition in some object module. This observation forms the basis of program linking. The linker processes all object modules being linked and builds a table of all public definitions and their load time addresses. Linking for `alpha` is simply a matter of searching for `alpha` in this table and copying its linked address into the word containing the external reference.

A *name table* (NTAB) is defined for use in program linking. Each entry of the table contains the following fields:

Symbol	:	symbolic name of an external reference or an object module.
Linked_address	:	For a public definition, this field contains linked address of the symbol. For an object module, it contains the linked origin of the object module.

Most information in NTAB is derived from LINKTAB entries with *type* = PD.

Algorithm 7.2 (Program Linking)

1. *program_linked_origin* := <*link origin*> from `linker` command.
2. For each object module

(a) *t_origin* := *translated origin* of the object module;
OM_size := *size* of the object module;

(b) *relocation_factor* := *program_linked_origin* − *t_origin*;

(c) Read the machine language program in *work_area*.

(d) Read LINKTAB of the object module.

(e) For each LINKTAB entry with *type* = PD
name := *symbol*;
linked_address := *translated_address* + *relocation_factor*;
Enter (*name*, *linked_address*) in NTAB.

(f) Enter (object module name, *program_linked_origin*) in NTAB.

(g) *program_linked_origin* := *program_linked_origin* + *OM_size*;

3. For each object module

(a) *t_origin* := translated origin of the object module;
program_linked_origin := *load_address* from NTAB;

(b) For each LINKTAB entry with *type* = EXT

(i) *address_in_work_area* := address of *work_area* +
program_linked_origin − <*link origin*>
+ *translated address* − *t_origin*;

(ii) Search *symbol* in NTAB and copy its linked address. Add the linked address to the operand address in the word with the address *address_in_work_area*.

Example 7.9 While linking program P of Ex. 7.2 and program Q of Ex. 7.5 with *linked_origin* = 900, NTAB contains the following information

symbol	*linked address*
P	900
A	940
Q	942
ALPHA	973

Let the address of *work_area* be 300. When the LINKTAB entry of ALPHA is processed during linking, *address_in_work area* := 300 + 900 − 900 + 518 − 500, i.e. 318. Hence, the linked address of ALPHA, i.e. 973, is copied from the NTAB entry of ALPHA and added to the word in address 318 (step 3(b)(ii)).

EXERCISE 7.2

1. It is required to merge a set of object modules $\{om_i\}$ to construct a single object module *om'*. This reduces the linking and relocation time in situations where the object modules in $\{om_i\}$ are interdependent—that is, they call or use each other.

(a) What are the public definitions of *om'*?

(b) What are the external references in *om'*?

(c) Explain how the RELOCTAB's and LINKTAB's can be merged.

2. Comment on the feasibility of reconstructing the original object modules from *om'*. Is any additional information required to facilitate it?

7.3 SELF-RELOCATING PROGRAMS

The manner in which a program can be modified, or can modify itself, to execute from a given load origin can be used to classify programs into the following:

1. Non relocatable programs,
2. Relocatable programs,
3. Self-relocating programs.

A non relocatable program is a program which cannot be executed in any memory area other than the area starting on its translated origin. Non relocatability is the result of address sensitivity of a program and lack of information concerning the address sensitive instructions in the program. The difference between a relocatable program and a non relocatable program is the availability of information concerning the address sensitive instructions in it. A relocatable program can be processed to relocate it to a desired area of memory. Representative examples of non relocatable and relocatable programs are a hand coded machine language program and an object module, respectively.

A self-relocating program is a program which can perform the relocation of its own address sensitive instructions. It contains the following two provisions for this purpose:

1. A table of information concerning the address sensitive instructions exists as a part of the program.
2. Code to perform the relocation of address sensitive instructions also exists as a part of the program. This is called the *relocating logic*.

The start address of the relocating logic is specified as the execution start address of the program. Thus the relocating logic gains control when the program is loaded in memory for execution. It uses the load address and the information concerning address sensitive instructions to perform its own relocation. Execution control is now transferred to the relocated program.

A self-relocating program can execute in any area of the memory. This is very important in time sharing operating systems where the load address of a program is likely to be different for different executions.

EXERCISE 7.3

1. Comment on the following statements:

   (a) Self-relocating programs are less efficient than relocatable programs.

(b) There would be no need for linkers if all programs are coded as self relocating programs.

2. A self-relocating program needs to find its load address before it can execute its relocating logic. Comment on how this information can be determined by the program.

7.4 A LINKER FOR MS DOS

We discuss the design of a linker for the Intel 8088/80x86 processors which resembles LINK of MS DOS in many respects. The design uses the schematics of the previous section. It may be noted that the object modules of MS DOS differ from the Intel specifications in some respects. Further simplifications are made for the purpose of this discussion.

Object Module Format

An Intel 8088 object module is a sequence of *object records*, each object record describing specific aspects of the programs in the object module. There are 14 types of object records containing the following five basic categories of information:

1. Binary image (i.e. code generated by a translator)
2. External references
3. Public definitions
4. Debugging information (e.g. line number in source program)
5. Miscellaneous information (e.g. comments in the source program).

We only consider the object records corresponding to first three categories—a total of eight object record types.

The names and purpose of the object record types is summarized in Table 7.1. The formats of the records are shown in Fig. 7.5. The object records of an object module must appear in the specific order shown in Table 7.1. Each object record contains variable length information and may refer to the contents of previous object records. Each name in an object record is represented in the following format:

length (1 byte)	name

THEADR, LNAMES and SEGDEF records

The module name in the THEADR record is typically derived by the translator from the source file name. This name is used by the linker to report errors. An assembly programmer can specify the module name in the NAME directive. The LNAMES record lists the names for use by SEGDEF records. A SEGDEF record designates a segment name using an index into this list. The *attributes* field of a SEGDEF record indicates whether the segment is relocatable or absolute, whether (and in what manner) it can be combined with other segments, as also the alignment requirement of its base address (e.g. byte, word or paragraph, i.e. 16 byte, alignment). Stack segments

THEADR record

80H	length	T-module name	check-sum

LNAMES record

96H	length	name list	check-sum

SEGDEF record

98H	length	attributes (1-4)	segment length (2)	name index (1)	check-sum

EXTDEF record

8CH	length	external reference list	check-sum

PUBDEF record

90H	length	base (2-4)	name	offset (2)	···	check-sum

LEDATA record

A0H	length	segment index (1-2)	data offset (2)	data	check-sum

FIXUPP record

9CH	length	locat (1)	fix dat (1)	frame datum (1)	target datum (1)	target offset (2)	···	check sum

MODEND record

8AH	length	type (1)	start addr (5)	check-sum

Fig. 7.5 Object record formats

Table 7.1 Object Records of Intel 8088

Record type	*Id (Hex)*	*Description*
THEADR	80	Translator header record
LNAMES	96	List of names record
SEGDEF	98	Segment definition record
EXTDEF	8C	External names definition record
PUBDEF	90	Public names definition record
LEDATA	AO	Enumerated data (binary image)
FIXUPP	9C	Fixup record
MODEND	8A	Module end record

with the same name are concatenated with each other, while common segments with the same name are overlapped with one another. The *attribute* field also contains the origin specification for an absolute segment.

EXTDEF and PUBDEF records

The EXTDEF record contains a list of external references used by the programs of this module. A FIXUPP record designates an external symbol name by using an index into this list. A PUBDEF record contains a list of public names declared in a segment of the object module. The *base specification* identifies the segment. Each (*name, offset*) pair in the record defines one public name, specifying the name of the symbol and its offset within the segment designated by the base specification.

LEDATA records

An LEDATA record contains the binary image of the code generated by the language translator. *segment index* identifies the segment to which the code belongs, and *offset* specifies the location of the code within the segment.

FIXUPP records

A FIXUPP record contains information for one or more relocation and linking fixups to be performed. The *locat* field contains a numeric code called *loc code* to indicate the type of a fixup. The meanings of these codes are given in Table 7.2.

Table 7.2 FIXUPP codes

Loc code	*Meaning*
0	low order byte is to be fixed
1	offset is to be fixed
2	segment is to be fixed
3	pointer (i.e., segment : offset) is to be fixed

locat also contains the offset of the fixup location in the previous LEDATA record. The *frame datum* field, which refers to a SEGDEF record, identifies the segment to which the fixup location belongs. The *target datum* and *target offset* fields specify the relocation or linking information. *target datum* contains a segment index or an external index, while *target offset* contains an offset from the name indicated in *target datum*. The *fix dat* field indicates the manner in which the *target datum* and *target offset* fields are to be interpreted. The numeric codes used for this purpose are given in Table 7.3.

Table 7.3 Codes in *fixdat* field

code	*contents of target datum and offset fields*
0	segment index and displacement
2	external index and target displacement
4	segment index (offset field is not used)
6	external index (offset field is not used)

Example 7.10 Consider the assembly program

```
EXTRN      ABC, XYZ
LEA        BX, ABC+25H
MOV        AX, OFFSET XYZ
- - -
```

In the LEA statement, the assembler can put zeroes in the second operand field of the instruction and use code '2' in *fix dat* to indicate the linking requirement. ABC would be identified through the external index put in *target datum*, while 25H would be put in the *target offset* field. *loc code* would be '3'. An alternative would have been to put 25H in the operand field of the instruction and simply use code '6' in *fix dat* and '2' in *loc code* to indicate the linking requirement. For the MOV statement code '6' would be used in *fix dat*. *locat* would contain '1' indicating that offset of the effective address is to be put in the designated location.

MODEND record

The MODEND record signifies the end of the module, with the *type* field indicating whether it is the main program. This record also optionally indicates the execution start address. This has two components: (a) the segment, designated as an index into the list of segment names defined in SEGDEF record(s), and (b) an offset within the segment. The exact format in which this information is represented is not important for our discussion.

Example 7.11 Figure 7.6 illustrates two assembly language programs, each consisting of a single segment, which are assembled separately. From the viewpoint of linking, only the two MOV statements of segment COMPUTE and the CALL, LEA and MOV statements of segment PART2 are of interest. Hence only the object module records concerning these statements are shown in Fig. 7.7. Note that the object modules have been given

```
Sr. no.                          Statement                          Offset
0001              NAME      FIRST
0002   COMPUTE    SEGMENT
0003              EXTRN     PHI:BYTE, PSI:WORD
0004              PUBLIC    ALPHA, BETA
0007   ALPHA      ...                                                0015
  ⋮                ⋮
0012              MOV       AX, SEG PHI                             0028
  ⋮                ⋮
0022              MOV       AX, OFFSET PSI                          0056
  ⋮                ⋮
0029   BETA       ...                                                0084
  ⋮                ⋮
0035              DB        25                                      0123
0036   COMPUTE    ENDS
0037              END


0001              NAME      SECOND
0002   PART2      SEGMENT   PARA
0003              EXTRN     ALPHA,BETA:FAR,GAMMA
0004              PUBLIC    PHI, PSI
0010              JMP       BETA                                    0018
  ⋮                ⋮
0017   PHI        ...                                                0033
  ⋮                ⋮
0027   PSI        ...                                                0059
  ⋮                ⋮
0051              LEA       BX, ALPHA+20H                           0245
  ⋮                ⋮
0057              MOV       AX, SEG GAMMA                           0279
  ⋮                ⋮
0069   PART2      ENDS
0070              END
```

Fig. 7.6 Sample MS DOS assembly language programs

Object Module FIRST

Type	*Length*	*Other fields*	*Check sum*	
80H	...	05 FIRST	...	THEADR
96H	...	07 COMPUTE	...	LNAMES
98H	...	20H 124 01	...	SEGDEF
90H	...	01 05 ALPHA 0015	...	PUBDEF
90H	...	01 04 BETA 0084	...	PUBDEF
8CH	...	03 PHI 03 PSI	...	EXTDEF
A0H	...	01 0028 A1 00 00	...	LEDATA
9CH	...	8801 06 01 01	...	FIXUPP
A0H	...	01 0056 A1 00 00	...	LEDATA
9CH	...	8401 06 01 02	...	FIXUPP
8AH	...	C0H 01 00	...	MODEND

Object Module SECOND

Type	*Length*	*Other fields*	*Check sum*	
80H	...	06 SECOND	...	THEADR
96H	...	05 PART2	...	LNAMES
98H	...	60H 398 01	...	SEGDEF
90H	...	01 03 PHI 0033	...	PUBDEF
90H	...	01 03 PSI 0059	...	PUBDEF
8CH	...	05 ALPHA 04 BETA 05 GAMMA	...	EXTDEF
A0H	...	01 0018 EA 00 00 00 00	...	LEDATA
9CH	...	8C01 06 01 02	...	FIXUPP
A0H	...	01 0245 8D 1E 00 00	...	LEDATA
9CH	...	C402 02 01 01 00 20H	...	FIXUPP
A0H	...	01 0279 A1 00 00	...	LEDATA
9CH	...	8801 06 01 03	...	FIXUPP
8AH	...	80H	...	MODEND

Fig. 7.7 MS DOS object modules

the names FIRST and SECOND through the NAME directive. For simplicity, we show addresses in decimal.

In the object module FIRST, the SEGDEF code of 20H indicates that the segment is byte aligned and relocatable. For both MOV statements involving the use of external symbols, the assembler generates identical code with zeroes in the second operand field. In the first FIXUPP record, 88 in the first byte of *locat* implies *loc code* = '2' indicating that fixing is to be performed by putting the segment base of the external symbol in the generated code. *fix dat* contains the code '6', indicating that an external symbol is involved in the fixup. The next two fields indicate that fixing is to be performed in the first segment of the module, according to the external symbol #1 (i.e., PHI). The second FIXUPP record is similar except that 84 in *locat* implies *loc code* = '1' (offset to be put in the target location) and *target datum* = '2' implying that fixing is to be performed according to external symbol and target displacement. (Note that

code '6' could have been used as discussed in Ex. 7.10). The next two fields indicate that fixing is to be performed in the first segment of the module according to the external symbol #2 (i.e. PSI). In SECOND, the JMP statement is fixed using the code 8C in *locat*, implying *loc code* = '3' which indicates that both segment base and offset are to be put into the generated code. The statement

```
LEA          BX, ALPHA+20H
```

is assumed to have been assembled with zeroes in the operand field. It is fixed by using the code C4 in *locat* (i.e. *loc* = 2) and *code* = 2 in *fix dat*. The displacement 20H appears in *target offset* field of the FIXUPP record.

7.4.1 Design of the Linker

We shall present the design of a program named LINKER which performs both linking and relocation. The output of LINKER is a binary program which resembles a program with .COM extension in MS DOS. It is assumed that the binary program cannot be relocated by the loader. (Note that LINK program of MS DOS produces a program with .EXE extension, which is relocated by the loader prior to execution. The reader would do well to note this difference.)

Specification

The LINKER invocation command has the following format:

LINKER <*object module names*>, <*executable file*>,
<*load origin*>, <*list of library files*>

LINKER performs relocation and linking of all named object modules to produce a binary program with the specified load origin. The program is stored in <*executable file*>. When LINKER comes across an external name ext_symb_i not defined in any of the object modules named in the LINKER command, it locates an object module om_i in <*list of library files*> which contains ext_symb_i as a public definition. om_i is now included in the set of object modules to be linked and relocated. This process of resolving an external reference by including the object module containing its definition from a library file is called *autolinking*. LINKER execution terminates when all external references have been resolved, or when the unresolved external references cannot be found in any of the library files.

Example 7.12 In the LINKER command

```
LINKER alpha+beta+min, calculate, 10000, pas.lib
```

alpha, beta and min are names of the object modules to be linked and calculate is the name to be given to the executable file generated by LINKER. The load origin of the executable program is 10,000. Any external names not defined in object modules alpha, beta and min are to be searched in the library pas.lib.

Data structures and algorithm

LINKER uses a two pass strategy. In the first pass, the object modules are processed to collect information concerning segments and public definitions. The second pass performs relocation and linking.

First pass

In the first pass, LINKER only processes the object records relevant for building NTAB.

Algorithm 7.3 (First pass of LINKER)

1. *program_linked_origin* := <*load origin*>; (Use a default value if <*load origin*> is not specified.)
2. Repeat step 3 for each object module to be linked.
3. Select an object module and process its object records.
   (a) If an LNAMES record, enter the names in NAMELIST.
   (b) If a SEGDEF record
      (i) *i* := *name index*; *segment_name* := NAMELIST [i];
      segment_addr := start address in *attributes* (if any);
      (ii) If an absolute segment, enter (*segment_name*, *segment_addr*) in NTAB.
      (iii) If the segment is relocatable and cannot be combined with other segments
         – Align the address contained in *program_linked_origin* on the next word or paragraph as indicated in the *attributes* field.
         – Enter (*segment_name*, *program_linked_origin*) in NTAB.
         – *program_linked_origin* := *program load origin* + *segment length*;
   (c) For each PUBDEF record
      (i) *i* := *base*; *segment_name* := NAMELIST [i];
      symbol := *name*;
      (ii) *segment_addr* := load address of *segment_name* in NTAB;
      (iii) *sym_addr* := *segment_addr* + *offset*;
      (iv) Enter (*symbol*, *sym_addr*) in NTAB.

Example 7.13 For the LINKER command

```
LINKER FIRST+SECOND, demo, 10000, math.lib
```

where object modules FIRST and SECOND are as shown in Ex. 7.11, NTAB at the end of first pass is as follows:

Symbol	*Load address*
COMPUTE	10000
ALPHA	10015
BETA	10084
PART2	10128
PHI	10161
PSI	10187

Note that PART2 has the load address of 10128 even though the first byte following the segment COMPUTE has the address 10124. This is because PART2 has a paragraph alignment.

Second pass

Second pass of LINKER constructs the executable program in *work_area*. Data from the LEDATA records is moved to appropriate parts of *work_area*. FIXUPP records are then processed to effect relocation and linking. At the end of the pass, the executable program is written into the current directory under the name <*executable file*> specified in the LINKER command.

It is simple to move the data from an LEDATA record into the appropriate part of *work_area* using *segment index* and *data offset*. The segment index is used to obtain the load origin of the segment from NTAB. Adding the data offset to it gives load address of the first byte of the data in the LEDATA record. Address of this byte in *work_area* is now obtained by a computation analogous to step 2(e)(ii) of Algorithm 7.1.

Some complications arise from the fact that *segment index* contains a numeric value rather than the name of a segment. The segment name can be obtained by using this value as an index into LNAMES record(s). This name can be searched in NTAB to obtain its load origin. Similarly, while processing a FIXUPP record, the target datum can be used as an index into EXTDEF or SEGDEF records to obtain the target name. This name can be searched in NTAB to obtain its load origin.

The second pass constructs a number of tables to eliminate indirect references to NTAB. To eliminate search in the LNAMES records, the NAMELIST table can be constructed once again in the second pass, and *segment index* can be used to index NAMELIST. However, the search in NTAB is still needed. To overcome this problem, LINKER builds another table called the *segment table* (SEGTAB) to contain all segment names defined in the object module. A SEGTAB entry has the following format:

segment name	*load address*

where the load address information is copied from NTAB. Now, while processing an LEDATA record, the segment index can be used to index SEGTAB to obtain

the segment's load address. A similar use can be made while processing FIXUPP records.

Since a linking specification in a FIXUPP record may contain a reference to an external symbol, LINKER builds an *external symbols table* (EXTTAB) with a similar entry-format, viz.

external symbol	*load address*

EXTTAB is built by processing the EXTDEF records and copying the load addresses from NTAB. (Actually the *name* fields of the SEGTAB and EXTTAB entries are redundant since access to an entry in these tables is through an index rather than through a name.)

When some external name `alpha` is not found in the set of object modules named in the LINKER command, LINKER performs autolinking. The object module containing `alpha` is included in the set of object modules being linked. To implement this, the first pass of LINKER is performed on the new object module. Processing of the FIXUPP record which triggered off autolinking for `alpha` is then resumed.

Algorithm 7.4 (Second pass of LINKER)

1. *list_of_object_modules* := Object modules named in LINKER command;
2. Repeat step 3 until *list_of_object_modules* is empty.
3. Select an object module and process its object records.
   (a) If an LNAMES record
       Enter the names in NAMELIST.
   (b) If a SEGDEF record
       i := *name index*; *segment_name* := NAMELIST [i];
       Enter (*segment_name*, *load address* from NTAB) in SEGTAB.
   (c) If an EXTDEF record
       (i) *external_name* := name from EXTDEF record;
       (ii) If *external_name* is not found in NTAB, then
           - Locate an object module in the library which contains *external_name* as a segment or public definition.
           - Add name of object module to *list_of_object_modules*.
           - Perform first pass of LINKER, i.e. Algorithm 7.3, for the new object module.
       (iii) Enter (*external_name*, *load address* from NTAB) in EXTTAB.
   (d) If an LEDATA record
       (i) *i* := *segment index*; *d* := *data offset*;
       (ii) *program_load_origin* := SEGTAB [i].*load address*;
       (iii) *address_in_work_area* := address of *work_area* +
               program_load_origin − <*load origin*> + *d*;

(iv) Move data from LEDATA into the memory area starting at the address *address_in_work_area*.

(e) If a FIXUPP record, for each FIXUPP specification

(i) f := *offset* from *locat* field;

(ii) *fix_up_address* := *address_in_work_area* + f;

(iii) Perform required fix up using a load address from SEGTAB or EXTTAB and the value of code in *locat* and *fix dat*.

(f) If a MODEND record

If start address is specified, compute the corresponding load address (analogous to the computation while processing an LEDATA record) and record it in the executable file being generated.

Example 7.14 Figure 7.8(a) illustrates the contents of the various data structures before object module `SECOND` is processed by the second pass of LINKER. While linking the object modules of Ex. 7.11 autolinking would be performed for the external name `GAMMA` of object module `SECOND`. Let `GAMMA` be a public definition in object module `THIRD`. First pass of LINKER is performed for `THIRD`. Thus, the segments and public symbols defined in `THIRD` are added to NTAB. This includes `PRECISE`, a segment name and `EMP_DATA`, another public definition (see Fig. 7.8(b)). The name `THIRD` is also added to the list of object modules. This ensures that the second pass would be performed for `THIRD`.

EXERCISE 7.4

1. Explain the purpose of the *segment index* field in an LEDATA record.
2. Answer problems 1 and 2 of Exercise 7.2 with respect to the Intel 8086 object module.
3. Modify Algorithm 7.4 to incorporate the processing of `common` and `stack` segments as follows:
   (a) All `common` segments with the same name overlap one another,
   (b) All `stack` segments with the same name are concatenated.
4. After a program has been translated to obtain the object module, it is found that certain n instructions from the translated address *aaaa* are redundant. These instructions can be replaced by *no-op* instructions, by introducing an LEDATA record with *aaaa* in its *data offset* field. However, this would leave the program length unchanged and also waste execution time due to the *no-op* instructions. It is therefore required to physically delete these n instructions from the object module.

   Indicate what modifications need to be made to implement physical deletion.
5. Suggest an entry format for the symbol table of an assembler which incorporates all information required to generate the SEGDEF, PUBDEF, EXTDEF and FIXUPP records of the object module.
6. Autolinking is a slow and laborious process because it involves library searches for every unresolved address constant. Comment on the effectiveness of the following schemes aimed at speeding up autolinking:

a) After processing FIRST

Symbol	*Load address*	
COMPUTE	10000	NTAB
ALPHA	10015	
BETA	10084	
PART2	10128	
PHI	10161	
PSI	10187	

Symbol	
COMPUTE	NAMELIST

Segment name	*Load address*	
COMPUTE	10000	SEGTAB

External symbol	*Load address*	
PHI	10161	EXTTAB
PSI	10187	

b) After processing SECOND

Symbol	*Load address*	
COMPUTE	10000	NTAB
ALPHA	10015	
BETA	10084	
PART2	10128	
PHI	10161	
PSI	10187	
PRECISE	10528	
GAMMA	10586	
EMP_DATA	10619	

Symbol	
PART2	NAMELIST

Segment name	*Load address*	
PART2	10128	SEGTAB

External symbol	*Load address*	
ALPHA	10015	EXTTAB
BETA	10084	
GAMMA	10586	

Fig. 7.8 LINKER data structures for autolinking

(a) A few unresolved address constants are grouped together for library searching.
(b) The directory entries for the object modules library are extended to include information about all public definitions in an object module.

7. In Section 7.4.1 we have assumed that *work_area* is large enough to accommodate the entire binary program. Comment on the changes required if *work_area* is smaller than the binary program.

7.5 LINKING FOR OVERLAYS

Definition 7.4 (Overlay) *An* overlay *is a part of a program (or software package) which has the same load origin as some other part(s) of the program.*

Overlays are used to reduce the main memory requirement of a program.

Overlay structured programs

We refer to a program containing overlays as an *overlay structured program*. Such a program consists of

1. A permanently resident portion, called the *root*
2. A set of overlays.

Execution of an overlay structured program proceeds as follows: To start with, the *root* is loaded in memory and given control for the purpose of execution. Other overlays are loaded as and when needed. Note that the loading of an overlay overwrites a previously loaded overlay with the same load origin. This reduces the memory requirement of a program. It also makes it possible to execute programs whose size exceeds the amount of memory which can be allocated to them.

The overlay structure of a program is designed by identifying mutually exclusive modules—that is, modules which do not call each other. Such modules do not need to reside simultaneously in memory. Hence they are located in different overlays with the same load origin.

Example 7.15 Consider a program with 6 sections named `init`, `read`, `trans_a`, `trans_b`, `trans_c`, and `print`. `init` performs some initializations and passes control to `read`. `read` reads one set of data and invokes one of `trans_a`, `trans_b` or `trans_c` depending on the values of the data. `print` is called to print the results.

`trans_a`, `trans_b` and `trans_c` are mutually exclusive. Hence they can be made into separate overlays. `read` and `print` are put in the root of the program since they are needed for each set of data. For simplicity, we put `init` also in the root, though it could be made into an overlay by itself. Figure 7.9 shows the proposed structure of the program. The overlay structured program can execute in 40 K bytes though it has a total size of 65 K bytes. It is possible to overlay parts of `trans_a` against each other by analyzing its logic. This will further reduce the memory requirements of the program.

The overlay structure of an object program is specified in the linker command.

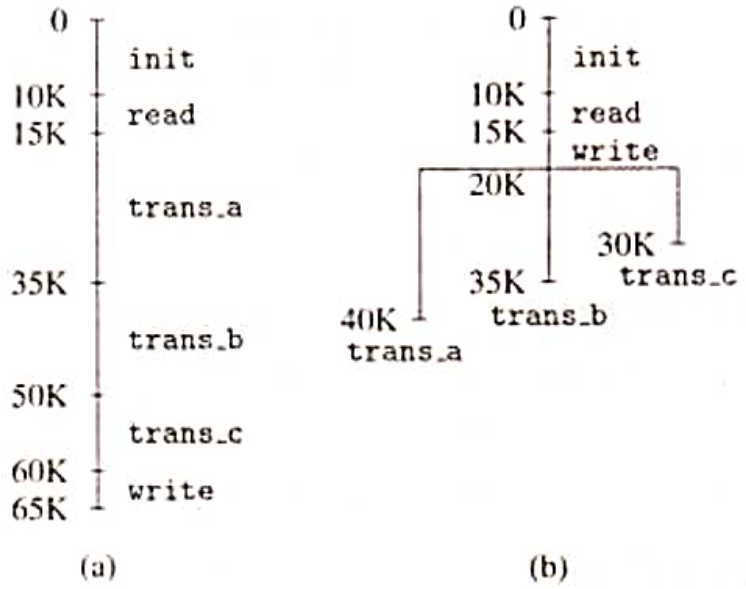


Fig. 7.9 An overlay tree

Example 7.16 The MS-DOS LINK command to implement the overlay structure of Fig. 7.9 is

```
LINK  init + read + write + (trans_a) + (trans_b)
      + (trans_c), <executable file>, <library files>
```

The object module(s) within parenthesis become one overlay of the program. The object modules not included in any overlay become part of the root. LINK produces a single binary program containing all overlays and stores it in *<executable file>*.

Example 7.17 The IBM mainframe linker commands for the overlay structure of Fig. 7.9 are as follows:

```
Phase main:      PHASE     MAIN,+10000
                 INCLUDE   INIT
                 INCLUDE   READ
                 INCLUDE   WRITE
Phase a_trans:   PHASE     A_TRANS,*
                 INCLUDE   TRANS_A
Phase b_trans:   PHASE     B_TRANS,A_TRANS
                 INCLUDE   TRANS_B
Phase c_trans:   PHASE     C_TRANS,A_TRANS
                 INCLUDE   TRANS_C
```

Each overlay forms a binary program (i.e. a *phase*) by itself. A PHASE statement specifies the object program name and linked origin for one overlay. The linked origin can be specified in many different ways. It can be an absolute address specification as in the first PHASE statement. In the second PHASE statement, the '*' indicates contents of the counter *program_linked_origin*. Thus, the linked origin is the next available memory byte. The specifications in the last two PHASE statements indicate the same linked origin as that of A_TRANS.

Execution of an overlay structured program

For linking and execution of an overlay structured program in MS DOS the linker produces a single executable file at the output, which contains two provisions to support overlays. First, an overlay manager module is included in the executable file. This module is responsible for loading the overlays when needed. Second, all calls that cross overlay boundaries are replaced by an interrupt producing instruction. To start with, the overlay manager receives control and loads the root. A procedure call which crosses overlay boundaries leads to an interrupt. This interrupt is processed by the overlay manager and the appropriate overlay is loaded into memory.

When each overlay is structured into a separate binary program, as in IBM mainframe systems, a call which crosses overlay boundaries leads to an interrupt which is attended by the OS kernel. Control is now transferred to the OS loader to load the appropriate binary program.

Changes in LINKER algorithms

The basic change required in LINKER algorithms of Section 7.4.1 is in the assignment of load addresses to segments. *program_load_origin* can be used as before while processing the root portion of a program. The size of the root would decide the load address of the overlays. *program_load_origin* would be initialized to this value while processing every overlay. Another change in the LINKER algorithm would be in the handling of procedure calls that cross overlay boundaries. LINKER has to identify an inter-overlay call and determine the destination overlay. This information must be encoded in the software interrupt instruction.

An open issue in the linking of overlay structured programs is the handling of object modules added during autolinking. Should these object modules be added to the current overlay or to the root of the program? The former has the advantage of cleaner semantics (think of procedures with *static* or *own* data), however it may increase the memory requirement of the program.

EXERCISE 7.5

1. A call matrix (CM) is a square matrix with boolean values in which CM[*i*, *j*] = *true* only if procedure *i* calls procedure *j* (directly or indirectly). Describe how a programmer can use this information to design the overlay structure of a program.
2. Modify the passes of LINKER to perform the linking of an overlay structured program. (*Hint:* Should each overlay have a separate NTAB?)
3. This chapter has discussed *static relocation*. *Dynamic relocation* implies relocation during the execution of a program, e.g. a program executing in one area of memory may be suspended, relocated to execute in another area of memory and resumed there. Discuss how dynamic relocation can be performed.
4. An inter-overlay call in an overlay structured program is expensive from the execution viewpoint. It is therefore proposed that if sufficient memory is available during program execution, all overlays can be loaded into memory so that each inter-overlay call can simply be executed as an inter-segment call. Comment on the feasibility of

this proposal and develop a design for it.

7.6 LOADERS

As described in Section 7.1.1, an absolute loader can only load programs with load origin = linked origin. This can be inconvenient if the load address of a program is likely to be different for different executions of a program. A *relocating loader* performs relocation while loading a program for execution. This permits a program to be executed in different parts of the memory.

Linking and loading in MS DOS

MS DOS operating system supports two object program forms. A file with a .COM extension contains a non relocatable object program whereas a file with a .EXE extension contains a relocatable program. MS DOS contains a program EXE2BIN which converts a .EXE program into a .COM program. The system contains two loaders—an absolute loader and a relocating loader. When the user types a filename with the .COM extension, the absolute loader is invoked to simply load the object program into the memory. When a filename with the .EXE extension is given, the relocating loader relocates the program to the designated load area before passing control to it for execution.

EXERCISE 7.6

1. Compare the .COM files in MS DOS with an object module. (MS DOS object modules are contained in files with .OBJ extension.)
2. It is proposed to perform dynamic linking and loading of the program units constituting an application. Comment on the nature of the information which would have to be maintained during the execution of a program.

BIBLIOGRAPHY

Most literature on loaders and linkage editors is proprietary. Some public domain literature is listed below.

1. Barron, D. W. (1969): *Assemblers and Loaders*, Macdonald Elsevier, London.
2. Presser, L. and J. R. White (1972): "Linkers and Loaders," *Computing Surveys*, **4** (3).
3. Wilder, W. L. (1980): "Comparing load and go and link/load compiler organizations," *Proc. AFIPS NCC*, **49**, 823-825.
4. Schwartz, R. L. (1978): "Parallel compilation: A design and its application to SIMULA-67," *Computer Languages*, **3** (2), 75-94.

CHAPTER 8

Software Tools

Computing involves two main activities—program development and use of application software. Language processors and operating systems play an obvious role in these activities. A less obvious but vital role is played by programs that help in developing and using other programs. These programs, called *software tools*, perform various housekeeping tasks involved in program development and application usage.

Definition 8.1 *A* software tool *is a system program which*

1. *interfaces a program with the entity generating its input data, or*
2. *interfaces the results of a program with the entity consuming them.*

The entity generating the data or consuming the results may be a program or a user. Figure 8.1 shows a schematic of a software tool.

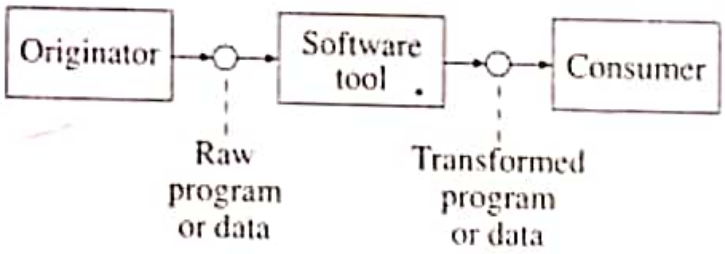


Fig. 8.1 A software tool

Example 8.1 A file rewriting utility organizes the data in a file in a format suitable for processing by a program. The utility may perform blocking/deblocking of data (see Section 16.5.2), padding and truncation of fields and records, sorting of records, etc. The file rewriting utility is a software tool according to Part 1 of Definition 8.1.

In this chapter we discuss two kinds of software tools—software tools for program development, and user interfaces.

8.1 SOFTWARE TOOLS FOR PROGRAM DEVELOPMENT

The fundamental steps in program development are:

1. Program design, coding and documentation
2. Preparation of programs in machine readable form
3. Program translation, linking and loading
4. Program testing and debugging
5. Performance tuning
6. Reformatting the data and/or results of a program to suit other programs.

Step 3 requires use of language processors. All other steps involve transformations of programs or data which fall within the purview of Definition 8.1. The software tools performing these transformations are described in the following Sections.

8.1.1 Program Design and Coding

Two categories of tools used in program design and coding are

1. Program generators
2. Programming environments.

As described in Section 1.2.1, a *program generator* generates a program which performs a set of functions described in its specification. Use of a program generator saves substantial design effort since a programmer merely specifies *what* functions a program should perform rather than *how* the functions should be implemented. Coding effort is saved since the program is generated rather than coded by hand. A *programming environment* (see Section 8.4) supports program coding by incorporating awareness of the programming language syntax and semantics in the language editor.

8.1.2 Program Entry and Editing

These tools are text editors or more sophisticated programs with text editors as front ends. The editor functions in two modes. In the *command mode*, it accepts user commands specifying the editing function to be performed. In the *data mode*, the user keys in the text to be added to the file. Failure to recognize the current mode of the editor can lead to mix up of commands and data. This can be avoided in two ways. In one approach, a quick exit is provided from the data mode, e.g. by pressing the `escape` key, such that the editor enters the command mode. The `vi` editor of Unix uses this approach. Another popular approach is to use the screen mode (also called the what-you-see-is-what-you-get mode), wherein the editor is in the data mode most of the time. The user is provided special keys to move the cursor on the screen. A stroke of any other key is taken to imply input of the corresponding character at the current cursor position. Certain keys pressed along with the `control`

key signify commands like erase character, delete line, etc. Thus end of data need not be explicitly indicated by the user. Most Turbo editors on PC's use this approach.

Editors are discussed in greater detail in Section 8.2.

8.1.3 Program Testing and Debugging

Important steps in program testing and debugging are selection of test data for the program, analysis of test results to detect errors (if any), and debugging, i.e. localization and removal of errors. Software tools to assist the programmer in these steps come in the following forms:

1. *Test data generators* help the user in selecting test data for his program. Their use helps in ensuring that a program is thoroughly tested.
2. *Automated test drivers* help in *regression testing*, wherein a program's correctness is verified by subjecting it to a standard set of tests after every modification. Regression testing is performed as follows: Many sets of test data are prepared for a program. These are given as inputs to the test driver (see Fig. 8.2). The driver selects one set of test data at a time and organizes execution of the program on the data.

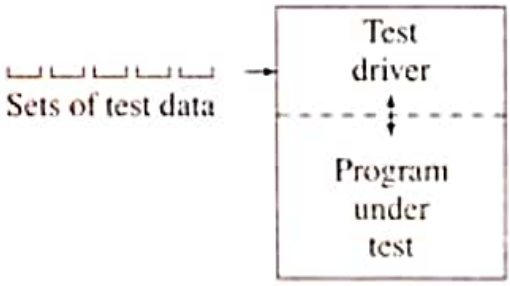


Fig. 8.2 Automated test driver

3. *Debug monitors* help in obtaining information for localization of errors.
4. *Source code control systems* help to keep track of modifications in the source code.

Test data selection uses the notion of an *execution path* which is a sequence of program statements visited during an execution. For testing, a program can be viewed as a set of execution paths. A test data generator determines the conditions which must be satisfied by the program's inputs for control to flow along a specific execution path. A test data is a set of input values which satisfy these conditions.

Example 8.2 The execution path involving the statement `a := a + 1.0;` in Fig. 8.3 is traversed during execution only if $x > z$. To test this path, the test data generator must select a value for y such that $x > z$, e.g. $y = z^2$.

Two problems exist in automated testing of programs. One concerns wastage of testing effort on infeasible paths, i.e. paths which are never followed during the

```
x := sqrt(y) + 2.5;
if x > z then
   a := a + 1.0;
else
   ...
```

Fig. 8.3 Program for Example 8.2

execution of a program. The second concerns testing of complex loop structures. It is best to leave both these aspects to manual testing. Thus automated testing can be used to show up a first batch of errors at low cost. This would free the programmer and system analyst to tackle more complex aspects of testing.

Example 8.3 In Fig. 8.3, if a statement `z := y + 10;` precedes `x := sqrt(y) + 2.5;` then the statement `a := a + 1.0;` lies along an infeasible path.

Producing debug information

Classically, localization and removal of errors has been aided by special purpose debug information. Such information can be produced statically by analysing the source program or dynamically during program execution. Statically produced debug information takes the form of cross reference listings, lists of undefined variables and unreachable statements, etc. All these are useful in determining the cause of a program malfunction. Techniques of data flow analysis (see Section 6.5.3.2) are employed to collect such information.

Example 8.4 The data flow concept of reaching definitions can be used to determine whether variable x may have a value when execution reaches statement 10 in the following program

```
No.      Statement
         ...
10.      sum := x + 10;
         ...
```

If no definitions of x reach statement 10, then x is surely undefined in statement 10. If some definition(s) reach statement 10, then x *may* have a value when control reaches the statement. Whether x is defined in a specific execution of the program would depend on how control flows during the execution.

Dynamically produced debug information takes the form of value dumps and execution traces produced during the execution of a program. This information helps to determine the execution path followed during an execution and the sequence of values assumed by a variable. Most programming languages provide facilities to produce dynamic debug information.

Example 8.5 Figure 8.4 illustrates the trace and dump facility supported by most Fortran compilers. `debug unit` specifies the kind of dynamic debug information desired,

and the file in which it is to be written. The `at` specification is used to enable or disable the production of `trace` and `display` data when control reaches a specified statement during program execution.

1. `debug unit` (<*file_number*>, <*list_of_options*>)

   Debug output is written in the file <*file_number*>.
   <*list_of_options*> can contain:

`trace`	:	Trace of labelled statements executed
`subtrace`	:	Trace of subprograms called
`init` (<*list*>)	:	Trace of assignments made to each variable in <*list*>
`subchk` (<*list*>)	:	Subscript range check: Report error if subscript in any reference to an array in <*list*> is out of bounds

2. `at` <*label*>

   Indicated actions are executed when statement bearing <*label*> is encountered during execution

`trace on`	:	Trace is enabled
`trace off`	:	Trace is disabled
`display` <*list*>	:	Values of variables in the list are written in the debug file

Fig. 8.4 Trace and dump facilities in Fortran

Use of trace and dump facilities is cumbersome due to the volume of information produced. To improve the effectiveness of debugging, it should be possible for the user to dynamically control the production of debug information, i.e. to dynamically specify the trace or dump actions. These facilities can be provided by the language compiler or interpreter, or by a language independent tool.

Example 8.6 Figure 8.5 illustrates the use of dynamic debugging facilities supported by many Basic interpreters. The programmer can set breakpoints at many statements using the `stop on` command. The system initiates a debug conversation with the programmer when control reaches any breakpoint during program execution. During a debug conversation, the programmer can display or change values of variables and set or remove breakpoints.

A debug monitor is a software which provides debugging support for a program. The debug monitor executes the program being debugged under its own control. This provides execution efficiency during debugging. It also enables the monitor to perform dynamically specified debugging actions. A debug monitor can be made language independent, in which case it can handle programs written in many languages. The dynamic debugging technique (DDT) of DEC-10 is a well known example of this approach. The working principles of debug monitors are discussed in Section 8.3.

1. Breakpoint and Dump facilities:
   (a) `stop on` <*list of labels*>
   Sets breakpoint(s) at labelled statement(s).
   (b) `dump at` <*label*> <*list of variables*>
   Dumps values of variables at given label
2. Interactive debugging facilities :

   These commands can be issued when program reaches a breakpoint

`display` <*list of variables*>	:	Displays values of variables
`set` <*var*> = <*exp*>	:	Assigns value of <*exp*> to <*var*>
`resume`	:	Resumes execution
`run`	:	Restarts execution

Fig. 8.5 Debugging facilities in Basic

8.1.4 Enhancement of Program Performance

Program efficiency depends on two factors—the efficiency of the algorithm and the efficiency of its coding. An optimizing compiler can improve efficiency of the code but it cannot improve efficiency of the algorithm. Only a program designer can improve efficiency of an algorithm by rewriting it. However, this is a time consuming process hence some help should be provided to improve its cost-effectiveness. For example, it is better to focus on only those sections of a program which consume a considerable amount of execution time. A performance tuning tool helps in identifying such parts. It is empirically observed that less than three percent of program code generally accounts for more than 50 percent of program execution time. This observation promises major economies of effort in improving a program.

Example 8.7 A program consists of three modules A, B and C. Sizes of the modules and execution times consumed by them in a typical execution as given in Table 8.1.

Table 8.1

Name	*# of statements*	*% of total execution time*
A	150	4.00
B	80	6.00
C	35	90.00

It is seen that module C, which is roughly 13 percent of the total program size, consumes 90 percent of the program execution time. Hence optimization of module C would result in optimization for 90 percent of program execution time at only 13 percent of the cost.

A *profile monitor* is a software tool that collects information regarding the execution behaviour of a program, e.g. the amount of execution time consumed by its modules, and presents it in the form of an *execution profile*. Using this information, the programmer can focus attention on the program sections consuming a significant amount of execution time. These sections can be improved either through code optimization (see Chapter 6), or through improvement of the algorithm.

Example 8.8 Figure 8.6 shows a sample execution profile. The column '# of executions' in the profile indicates the number of times a statement was executed. '# of true' indicates how many times the condition in an `if` statement was found to be true. '% time' gives the percentage of execution time spent in executing a statement, the body of a loop, or a module.

```
Program alpha execution profile

program name   % execution time

main program        11.15
    sub1            41.59
    sub2            15.54
    meor            31.72

Program 'meor' consumed 31.72% of total time

        statement                      # of        %     # of
                                    executions   time   true

    subroutine meor(eoper)             5        31.72
    integer eoper(10)                  0
    k = 9                              5         0.08
    do 10 i1 = 1,10                   39        30.74
    j = i1 + k                        39         4.35
    if(sw.eq.1) goto 50               39         9.53     3
    if(eoper(j).eq.bl) goto 15        36         4.44    26
    if(eoper(j).eq.zero) goto 20      10         2.06     4
    pr = eoper(j)                      6         0.21
    goto 10                            6         0.21
15  ...
    ...
10  continue                          39         2.01
    ...
    end                                5         0.59
```

Fig. 8.6 An execution profile

The profile indicates that subroutine `meor` consumed 31.72 percent of the total execution time. The DO loop in `meor` consumed 30.74 percent of the time consumed by subroutine `meor`. This shows that it is important to optimize the loop. It is further seen

that the three `if` statements consume more than half of the loop execution time. It is possible to interchange the second and third `if` statements, however this is counter-productive since the second `if` is mostly true while the third `if` is mostly false (see '# of true' column). Hence there appears to be little scope for optimization of `meor` short of replacing the algorithm by a better one. An analysis of the profile would reveal this information in a couple of minutes, thereby saving considerable time and effort.

Information regarding execution counts, etc. is obtained by introducing counters in the program or in its generated code. This can be done by the language compiler as in IITFORT [Dhamdhere *et al*, 1979] or by a preprocessor [Ingalls, 1972].

8.1.5 Program Documentation

Most programming projects suffer from lack of up-to-date documentation. Automatic documentation tools are motivated by the desire to overcome this deficiency. These tools work on the source program to produce different forms of documentation, e.g. flow charts, IO specifications showing files and their records, etc.

8.1.6 Design of Software Tools

Program preprocessing and instrumentation

Program preprocessing techniques are used to support static analysis of programs. Tools generating cross reference listings and lists of unreferenced symbols; test data generators, and documentation aids use this technique. *Program instrumentation* implies insertion of statements in a program. The instrumented program is translated using a standard translator. During execution, the inserted statements perform a set of desired functions. Profile and debug monitors typically use this technique. In a profile monitor, an inserted statement updates a counter indicating the number of times a statement is executed, whereas in debug monitors an inserted statement indicates that execution has reached a specific point in the source program.

Example 8.9 A debug monitor instruments a program to insert statements of the form

```
call debug_mon (const_i);
```

before every statement in the program, where `const_i` is an integer constant indicating the serial number of the statement in the program. During execution of the instrumented program, the debug monitor receives control after every statement. Assume that the user has specified the following debug actions:

```
at 10, display total
at 20, display term
```

Every time debug monitor receives control, it checks to see if statement 10 or 20 is about to be executed. It then performs the debug action indicated by the user.

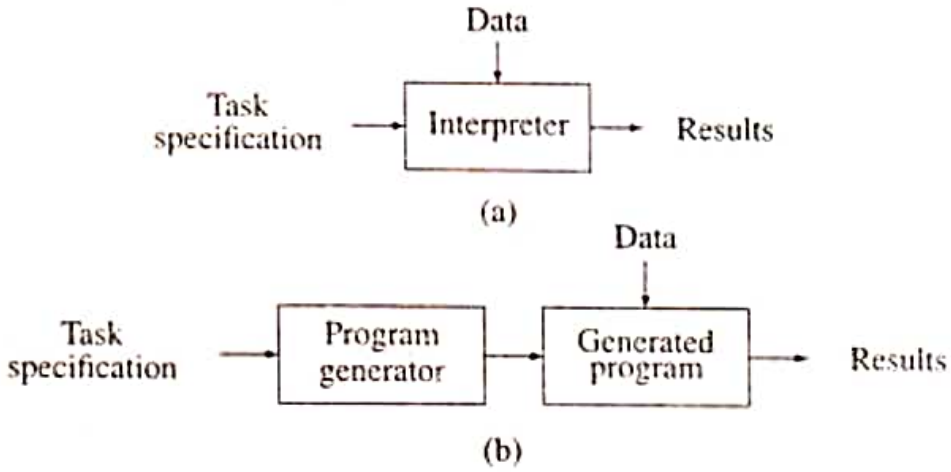


Fig. 8.7 Software tools using interpretation and program generation

Program interpretation and program generation

Figure 8.7 shows the schematic of software tools using the techniques of interpretation and program generation. Use of interpreters in software tools is motivated by the same reasons that motivate the use of interpreters in program development, viz. absence of a translation phase in processing a program. Since most requirements met by software tools are ad hoc, it is useful to eliminate the translation phase. However, interpreter based tools suffer from poor efficiency and poor portability, since an interpreter based tool is only as portable as the interpreter it uses. A generated program is more efficient and can be made portable.

Example 8.10 An organization specializing in the development of interactive applications needs to design user friendly screens for form fillin functions (see Ex. 1.2). It has been decided to use a software tool to obviate repeated coding efforts.

Use of the interpretive approach involves development of a general purpose interpreter for screen handling (we will call it the *screen handler*). Every application that needs to use the tool will have to interface with the screen handler. Whenever the application needs to use a screen, it will have to call the screen handler with the specification of the screen. If the program generator schematic is used, specification of the screen will have to be input to the program generator. The generated program will be in the form of a function which will become a part of the application.

From Fig. 8.7 it can be seen that the interpreter, besides being slow in execution, will require more memory. It will, however, permit dynamic changes in the specifications. The program generator provides execution efficiency. It also provides code space efficiency because only a routine tailored to a desired screen format needs to exist in the application. Any change in the specification of a screen will now imply generation of a new function and its integration with the application.

8.2 EDITORS

Text editors come in the following forms:

1. Line editors
2. Steam editors
3. Screen editors
4. Word processors
5. Structure editors

The scope of edit operations in a line editor is limited to a line of text. The line is designated *positionally*, e.g. by specifying its serial number in the text, or *contextually*, e.g. by specifying a context which uniquely identifies it. The primary advantage of line editors is their simplicity. A stream editor views the entire text as a stream of characters. This permits edit operations to cross line boundaries. Stream editors typically support character, line and context oriented commands based on the current editing context indicated by the position of a *text pointer*. The pointer can be manipulated using positioning or search commands.

Line and steam editors typically maintain multiple representations of text. One representation (the *display form*) shows the text as a sequence of lines. The editor maintains an *internal form* which is used to perform the edit operations. This form contains end-of-line characters and other edit characters. The editor ensures that these representations are compatible at every moment.

8.2.1 Screen editors

A line or stream editor does not display the text in the manner it would appear if printed. A screen editor uses the what-you-see-is-what-you-get principle in editor design. The editor displays a screenful of text at a time. The user can move the cursor over the screen, position it at the point where he desires to perform some editing and proceed with the editing directly. Thus it is possible to see the effect of an edit operation on the screen. This is very useful while formatting the text to produce printed documents.

8.2.2 Word Processors

Word processors are basically document editors with additional features to produce well formatted hard copy output. Essential features of word processors are commands for moving sections of text from one place to another, merging of text, and searching and replacement of words. Many word processors support a spell-check option. With the advent of personal computers, word processors have seen widespread use amongst authors, office personnel and computer professionals. Wordstar is a popular editor of this class.

8.2.3 Structure Editors

A structure editor incorporates an awareness of the structure of a document. This is useful in browsing through a document, e.g. if a programmer wishes to edit a specific

function in a program file. The structure is specified by the user while creating or modifying the document. Editing requirements are specified using the structure. A special class of structure editors, called syntax directed editors, are used in programming environments.

Example 8.11 NLS (short form for 'on line system') is an early structure editor [Engelbart, English 1968] oriented towards document editing. A document has a hierarchic structure, with different levels like group, plex, branch and statement. Items within a higher level item are siblings (i.e. brothers) with predecessor-successor relationships. Thus, one can talk of a group, the preceding or succeeding group, first statement in a group, etc.

Contemporary editors support a combination of line, string and screen editing functions. This makes it hard to classify them into the categories defined in this section. The `vi` editor of Unix and the editors in desk top publishing systems are typical examples of these.

8.2.4 Design of an Editor

The fundamental functions in editing are travelling, editing, viewing and display. Travelling implies movement of the editing context to a new position within the text. This may be done explicitly by the user (e.g. the line number command of a line editor) or may be implied in a user command (e.g. the search command of a stream editor). Viewing implies formatting the text in a manner desired by the user. This is an abstract view, independent of the physical characteristics of an IO device. The display component maps this view into the physical characteristics of the display device being used. This determines where a particular view may appear on the user's screen. The separation of viewing and display functions gives rise to interesting possibilities like multiple windows on the same screen, concurrent edit operations using the same display terminal, etc. A simple text editor may choose to combine the viewing and display functions.

Figure 8.8 illustrates the schematic of a simple editor. For a given position of the editing context, the editing and viewing filters operate on the internal form of text to prepare the forms suitable for editing and viewing. These forms are put in the editing and viewing buffers respectively. The viewing-and-display manager makes provision for appropriate display of this text. When the cursor position changes, the filters operate on a new portion of text to update the contents of the buffers. As editing is performed, the editing filter reflects the changes into the internal form and updates the contents of the viewing buffer.

Apart from the fundamental editing functions, most editors support an undo function to nullify one or more of the previous edit operations performed by the user. The undo function can be implemented by storing a stack of previous views or by devising an inverse for each edit operation. Multilevel undo commands pose obvious difficulties in implementing overlapping edits.

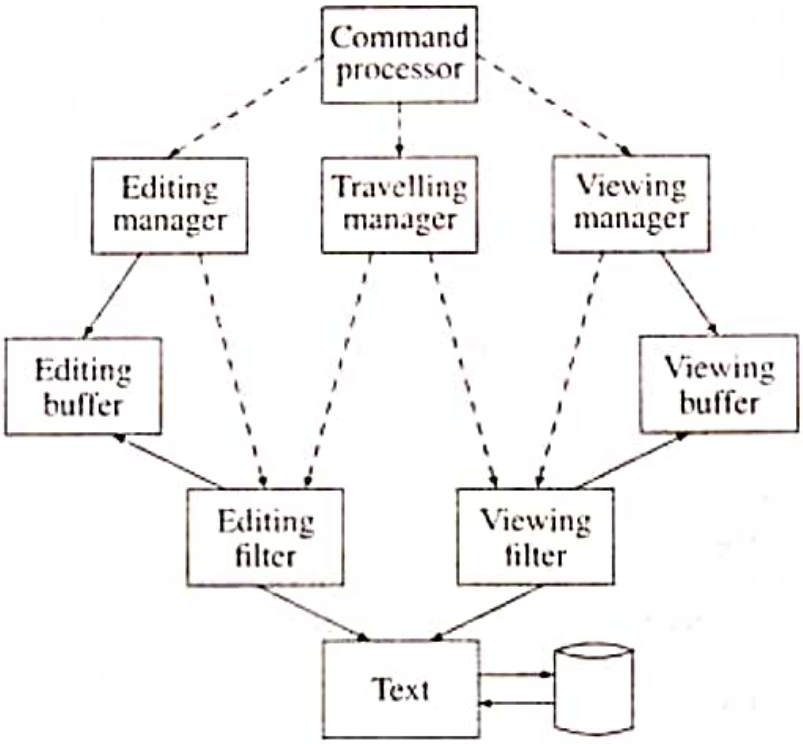


Fig. 8.8 Editor structure

8.3 DEBUG MONITORS

Debug monitors provide the following facilities for dynamic debugging:

1. Setting breakpoints in the program
2. Initiating a debug conversation when control reaches a breakpoint
3. Displaying values of variables
4. Assigning new values to variables
5. Testing user defined assertions and predicates involving program variables.

The debug monitor functions can be easily implemented in an interpreter. However interpretation incurs considerable execution time penalties. Debug monitors therefore rely on instrumentation of a compiled program to implement its functions. To enable the use of a debug monitor, the user must compile the program under the `debug` option. The compiler now inserts the following instructions:

A few `no-op` instructions of the form
`no-op` <*statement no*>

before each statement, where <*statement no*> is a constant indicating the serial number of the statement in the program. The compiler also generates a table containing the pairs (variable name, address). When a user gives a command to set a breakpoint at, say, statement 100, the debug monitor instruments the program to introduce the instruction

<*SI_instrn*><*Code*>

in place of the `no-op` instructions preceding `no-op 100`.

The compiled code for the program executes directly on the CPU until it reaches an <*SI_instrn*>. Execution of the <*SI_instrn*> produces a software interrupt with the interrupt code <*Code*>. This signifies a debug interrupt. (Alternatively the debug monitor can use a `BC ANY, DEBUG_MON` instruction instead of <*SI_instrn*>). The debug monitor now gains control and opens a debug conversation. The user may ask for display or modification of program variables. This is implemented using the (variable name, address) information produced by the compiler.

The sequence of steps involved in dynamic debugging of a program is as follows:

1. The user compiles the program under the `debug` option. The compiler produces two files—the compiled code file and the debug information file.
2. The user activates the debug monitor and indicates the name of the program to be debugged. The debug monitor opens the compiled code and debug information files for the program.
3. The user specifies his debug requirements—a list of breakpoints and actions to be performed at breakpoints. The debug monitor instruments the program, and builds a *debug table* containing the pairs (statement number, debug action).
4. The instrumented program gets control and executes up to a breakpoint.
5. A software interrupt is generated when the <*SI_instrn*> is executed. Control is given to the debug monitor which consults the debug table and performs the debug actions specified for the breakpoint. A debug conversation is now opened during which the user may issue some debug commands (which are implemented through interpretation) or modify breakpoints and debug actions associated with breakpoints. Control now returns to the instrumented program.
6. Steps 4 and 5 are repeated until the end of the debug session.

Unix supports two debuggers—`sdb` which is a PL level debugger and `adb` which is an assembly language debugger. `debug` of IBM PC is an object code level debugger.

8.3.1 Testing Assertions

A debug assertion is a relation between the values of program variables. An assertion can be associated with a program statement. The debug monitor verifies the assertion when execution reaches that statement. Program execution continues if the assertion is fulfilled, else a debug conversation is opened. The user can now perform actions to locate the cause of the program malfunction. Use of debug assertions eliminates the need to produce voluminous information for debugging purposes.

8.4 PROGRAMMING ENVIRONMENTS

A programming environment is a software : ystem that provides integrated facilities for program creation, editing, execution, testing and debugging. It consists of the following components:

1. A *syntax directed editor* (which is a structure editor)
2. A language processor—a compiler, interpreter, or both
3. A debug monitor
4. A dialog monitor.

All components are accessed through the dialog monitor. The syntax directed editor incorporates a front end for the programming language. As a user keys in his program, the editor performs syntax analysis and converts it into an intermediate representation (IR), typically an abstract syntax tree. The compiler (or interpreter) and the debug monitor share the IR. If a compiler is used, it is activated after the editor has converted a statement to IR. The compiler works incrementally to generate code for the statement. Thus, program execution or interpretation can be supported immediately after the last statement has been input. At any time during execution the programmer can interrupt program execution and enter the debug mode or return to the editor. In the latter case he can modify the program and resume or restart its execution.

The main simplification for the user is the easy accessibility of all functions through the dialog monitor. The system may also provide other program development and testing functions. For example, it may permit a programmer to execute a partially completed program. The programmer can be alerted if an undeclared variable or an incomplete statement is encountered during execution. The programmer can insert necessary declarations or statements and resume execution. This permits major interfaces in the program to be tested prior to the development of a module. Some programming environments also support reversible execution, whereby a program's execution can be 'stepped back' by one or more statements.

The Cornell Program Synthesizer

The Cornell Program Synthesizer (CPS) [Teitelman, Reps, 1981] is a syntax directed programming environment for a subset of PL/I known as PL/CS. It contains a syntax directed editor, a compilation-and-interpretation schematic for program execution, and a collection of debugging tools.

The editor guides the user to develop a well structured program through the use of templates. A *template* is a unit of information concerning PL/CS. It consists of some fixed syntax information and a few placeholders. A *placeholder* signifies a position in the program where a specific language construct, e.g. a statement or an expression, may be inserted by the programmer. (It is equivalent to a nonterminal symbol in a sentential form.) A program development session starts when the user keys in the command `main`. The editor now displays the template for a program, viz.

```
/* comment */
name :  procedure options(main);
  {declaration}
   {statement}
   END name;
```

where □ marks the position of the cursor, and { declaration } and { statement } are placeholders. The user can bring the cursor to a placeholder and begin to expand it by indicating the kind of entity he wishes to put there. CPS now replaces the placeholder by the template for that entity. Use of templates avoids syntax errors. Thus, errors like more **else**'s than **if**'s or missing **end**'s for **do** statements cannot arise in a CPS generated program.

Example 8.12 A user wishes to place an **if** statement in the placeholder { statement } in the initial template. When he indicates this, the program becomes

```
/* comment */
name :  procedure options(main);
   {declaration}
   if {condition} then {statement}
                  else {statement}
   END name;
```

The user can move the cursor to the new placeholder {statement}, indicate that an assignment statement is to be inserted in its place and type `a = b;` followed by the `return` key. The placeholder { statement } is now replaced by the statement `a = b;`. Moving the cursor to { statement } in the **else** part and pressing `return` indicates that the **else** clause is empty.

As the user keys in program statements, they are converted into an IR. Cursor position on the screen is mapped into the correct node of IR to support expansion of a placeholder. CPS supports execution of a partially complete program. When an unexpanded placeholder is encountered during execution, the user is prompted to expand it before resuming execution. The screen is split into two during execution. The lower half is used to display execution results and debug output, if any. The upper half displays a part of the user program in which the statement under execution is marked with the cursor symbol □. Thus, as execution proceeds, the flow of execution through the program is displayed on the screen. The user can interrupt execution at any moment and open a debug conversation or modify the program before resuming or restarting execution.

Experience with the use of CPS has uncovered some significant strengths and weaknesses. The template governed approach guarantees that every CPS generated program is syntactically valid. However, it makes certain operations, like insertion of a loop in an existing program, quite difficult. Another significant drawback is the absence of an undo facility. Later programming environments have removed some of these deficiencies. For example, the programming environment Poe offers unlimited

undo facilities. It also offers on line help. The user can query the various expansion options available at a placeholder and the scope and accessibility of variables at a given point in the program. All these facilities further enhance the utility of programming environments.

EXERCISE 8.4

1. Comment on the statement "Dynamic debugging is easier to implement in interpreters than in compilers".
2. Commeat on whether you would prefer a generative schematic or an interpretive schematic for following purposes:
   (a) Implementing `display` commands issued during dynamic debugging
   (b) Producing a report from a file
   (c) Writing a general purpose screen handling system
   (d) Handling data base queries.

   Give reasons for your answers.
3. A program profiling package is to be implemented for a Pascal compiler. The following options are being considered for its design:
   (a) Write a preprocessor which would introduce Pascal statements in a program to collect information during execution of the program,
   (b) Support a `profile` option in the compiler to perform program instrumentation.

   Which option would you use? Why?

8.5 USER INTERFACES

A *user interface* (UI) plays a vital role in simplifying the interaction of a user with an application. Classically, UI functionalities have two important aspects—issuing of commands and exchange of data. In early days of computing, a user was often the application designer or developer. Hence an understanding of commands and data was implicit in the use of an application. In those days UI's did not have an independent identity. This situation changed because of two reasons. First, as applications became larger a user was no longer expected to know all details concerning an application. Hence presentation of commands and prompts for data became important. Second, as applications grew to newer fields, it became necessary to assume a lower level of computer skills in an application user. This increased the importance of UI's by adding a new aspect, viz. on line help, to their functionality. The on line help component of the UI serves the function of educating the user in the capabilities of an application and the modalities of its usage. A well designed UI can make the difference between a grudging set of users and an excited user population (and even a fan following!) for an application.

A UI can be visualized to consist of two components—a *dialog manager* and a *presentation manager*. The dialog manager manages the conversation between the user and the application. This involves prompting the user for a command and

transmitting the command to the application. The presentation manager displays the data produced by the application in an appropriate manner on the user's display or printer device.

8.5.1 Command Dialogs

Commands are issued to an application through a command dialog. Three ways to implement command dialogs are:

1. Command languages
2. Command menus
3. Direct manipulation.

Command languages for computer applications are similar to command languages for operating systems. Primitive command languages support imperative commands with the syntax *<action> <parameters>*. More sophisticated command languages have both declarative and imperative commands and a syntax and semantics of their own. A practical difficulty in the use of a command language is the need to learn it before using the application. This implies a large commitment of time and effort on the part of the user, which makes casual use of the application forbidding. On line help can provide some relief by avoiding the need to memorize the syntax of commands, however it cannot eliminate the need to invest time and effort in initial learning of the command language.

Command menus provide obvious advantages to the casual user of an application, as the basic functionalities of the application are reflected in the menu choices. A hierarchy of menus can be used to guide the user into the details concerning a functionality. Interesting variations like pull-down menus are designed to simplify the use of menu systems. Most Turbo compilers use pull down command menus.

A *direct manipulation system* provides the user with a visual display of the universe of the application. The display shows the important objects in the universe. Actions or operations over objects are indicated using some kind of pointing device, e.g. a cursor or a mouse.

Example 8.13 `Lotus 1-2-3` is an example of a direct manipulation system. The universe of the application is the spreadsheet. The user can select a specific cell in the spreadsheet, change its contents and study the effect of this change on other cells.

Examples of direct manipulation can also be found in screen editors and video games.

Principles of command dialog design

Psychologists and human factors engineers have formulated a set of principles to ensure the effectiveness of command dialogs. Some of these are:

1. Ease of use

2. Consistency in command structure
3. Immediate feedback on user commands
4. Error handling
5. On line help to avoid memorizing command details
6. Undo facility
7. Shortcuts for experienced users.

Command menus score over command languages on the basis of principles 1, 2 and 5, while command languages are superior on the basis of principle 7. Some menu systems provide a *type ahead* facility while entering menu choices. This permits the user of a hierarchical menu system to type the menu choices for menus which are yet to be displayed on the user screen. This improves the appeal of the menu system for experienced users. Recent trends in command dialog design have been towards the use of direct manipulation systems with a graphics capability.

8.5.2 Presentation of Data

Data for an application can be input through free form typing. Alternatively, a form fillin approach may be used (see Ex. 1.2) when large volume of data is involved. Application results can be presented in the form of tables. Summary data can be presented in the form of graphs, pie charts, etc.

8.5.3 On Line Help

On line help is very important to promote and sustain interest in the use of an application. It minimizes the effort of initial learning of commands and avoids the distraction of having to consult a printed document to resolve every doubt. On line help to the users can be organized in the form of on line explanations, demonstrations, tutorials or on line manuals concerning commands. The on line help facility should be organized such that desired information can be efficiently located using the structure of the information, e.g. by searching for section headings, subtitles, figure or table names, footnotes, etc. Another effective method of organizing on line help is to provide context sensitive help, whereby a query can fetch different responses depending on the current position of the user in the application.

Hypertext

Hypertext visualizes a document to consist of a hierarchical arrangement of information units, and provides a variety of means to locate the required information. This takes the form of

1. Tables and indexes
2. String searching functions
3. Means to navigate within the document
4. Backtracking facilities.

Effectiveness of a hypertext document depends on the care with which it is organized. Hypertext authoring systems have been developed to help the application designer in the design of hypertext documents.

8.5.4 Structure of a User Interface

Figure 8.9 shows a UI schematic using a standard graphics package. The UI consists of two main components, presentation manager and dialog manager. The *presentation manager* is responsible for managing the user's screen and for accepting data and presenting results. The *dialog manager* is responsible for interpreting user commands and implementing them by invoking different modules of the application code. The dialog manager is also responsible for error messages and on line help functions, and for organizing changes in the visual context of the user.

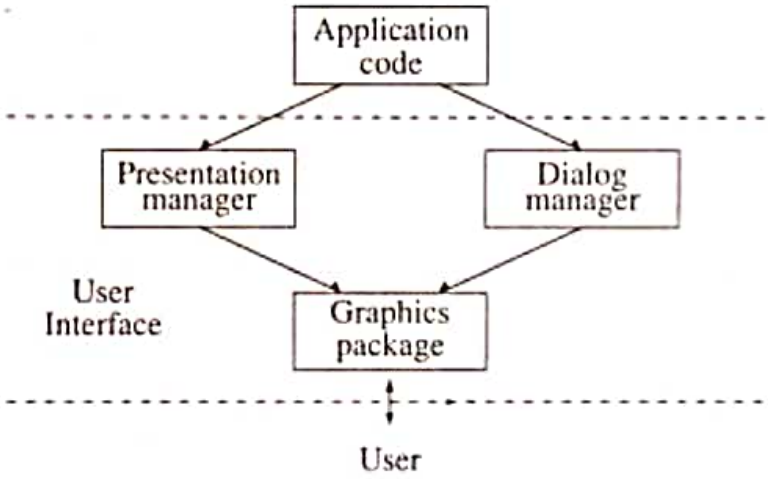


Fig. 8.9 User Interface

Example 8.14 The fields Employee name and Married in the form fillin screen of Fig. 1.8 (see Ex. 1.2) can be displayed by issuing the following commands to the graphics package:

```
Display ((2,5) Employee name :)
Drawbox ((2,25)(55,1))
Display ((10,5) Married :)
Drawdoublebox ((10,25)(2,1)(yes))
```

The dialog monitor is trivial in this application.

8.5.5 User Interface Management Systems

A *user interface management system* (UIMS) automates the generation of user interfaces. The UIMS accepts specification of the presentation and dialog semantics to produce the presentation and dialog managers of the UI respectively. The presentation and data managers could be generated programs specific to the presentation

and dialog semantics. Alternatively, the presentation and dialog managers could use interpretive schematics to implement the presentation and dialog semantics.

A variety of formalisms have been used to describe dialog semantics. These include grammars, event descriptions and finite state machines. In a grammar based description, the syntax and semantics of commands are specified in a YACC like manner. Thus, the interface is activated when a user types in a command. The event based approach uses a visual model of the interface. A screen with icons is displayed to the user. Selection of an icon by clicking the mouse on it causes an event. The action specified against the event is now performed.

The grammar and event description approaches lack the notion of a sequence of actions. The finite state machine approach can efficiently incorporate this notion. The basic principle in this approach is to associate a finite state machine with each window or each icon. Actions are specified on the basis of conditions involving the states of these machines. This approach has the additional power of coordinating the concurrent activities in different windows. In the following we describe two UIMSs using the event description approach.

Menulay

Menulay is an early UIMS using the screen layout as the basis for the dialog model. The UI designer starts by designing the user screen to consist of a set of icons. A semantic action is specified for each icon. This action is performed when the icon is selected. The interface consists of a set of screens. The system generates a set of icon tables giving the name and description of an icon, and a list of (event, function_id) pairs indicating the application function to be called when an event is selected.

Hypercard

This UIMS from Apple incorporates object orientation in the event oriented approach. A *card* has an associated screen layout containing buttons and fields. A button can be selected by clicking the mouse on it. A field contains editable text. Each card has a specific background, which itself behaves like a card. Many cards can share the same background. A hypercard program is thus a hierarchy of cards called a *stack*. UI behaviour is specified by associating an action, in the form of a HyperTalk script, with each button, field and card. The action for an event is determined by using the hierarchy of cards as an inheritance hierarchy. Hypercard uses an interpretive schematic to implement a UI.

EXERCISE 8.5

1. A library information system is to provide three functions—
   (a) Query the issue status of a book
   (b) Search the catalog by author and title
   (c) Issue or return a book.

Design a set of screens for these functions. Visualize the design of a UI using a Menulay or Hypercard like system.

BIBLIOGRAPHY

Kernighan and Plauger (1981) discuss various software tools based on the program preprocessing approach. Miller (1979) is a tutorial on software tools. Text editors are widely discussed in the literature. Meyrowitz and Dam (1981) is a comprehensive survey article on text editors. Myers (1979), Miller and Howden (1981) and Beizer (1983) are important sources on various aspects of program testing. Myers (1979) also describes software tools for program testing. Huang (1978) discusses fundamentals of program instrumentation for software testing. Ingalls (1972), Arthur and Ramanathan (1981) and Power (1983) describe use of instrumentation in different kinds of program analysis. Howden (1975), Koster (1980), Voges (1980) and Woodward and Hedley (1980) discuss aspects of test data generation. Prywes *et al* (1979) is an important article on automatic program generation. Martin (1982) describes generation of application programs. Allison (1983) discusses syntax directed editing. Barstow (1983) and Hunke (1981) are important sources on integrated programming environments.

(a) Software tools in general

1. Kernighan, B.W. and P.J.Plauger (1981): *Software Tools in Pascal*, Addison-Wesley, Reading.
2. Miller, E.(ed.) (1979): *Tutorial: Automated Tools for Software Engineering*, IEEE Computer Society Press.
3. Reifer, D.J. and S. Trattner (1977): "A glossary of software tools and techniques," *Computer*, **10** (7), 52-61.
4. Rochkind, M.J. (1975): "The source code control system," *IEEE Transactions on Software Engineering*, **1** (4), 364-370.

(b) Text editors

1. Embley, D.W. and G. Nagy (1981): "Behavioral aspects of text editors," *Computing Surveys*, **13** (1), 33-70.
2. Engelbart, D.C. and W.K. English (1968): "A research centre for augmenting human intellect," *AFIPS SJCC*, 33, 395-410.
3. Fraser, C.W.(1980): "A generalised text editor," *Commn. of ACM*, **23** (1), 27-60.
4. Macleod, I.A. (1977): "Design and implementation of a display oriented text editor," *Software – Practice and Experience*, **7** (6), 771-778.
5. Meyrowitz, N. and A. Van Dam (1982): "Interactive editing systems," Parts I and II, *Computing Surveys*, **14** (3), 321-352 and 353-416.

(c) Program testing and debugging

1. Beizer, B. (1983): *Software Testing Techniques*, Van Nostrand, New York, 1983.
2. Fairly, R.E.(1979):"ALADDIN: Assembly language assertion driven debug interpreter," *IEEE Transactions on Software Engineering*, **5** (4), 426-428.

3. Foster, K.A.(1980): "Error sensitive test cases analysis," *IEEE Transactions on Software Engineering*, **6**, 258-265.
4. Howden, W.F. (1975): "Methodology for generation of software test data," *IEEE Transactions on Computers*, **24** (5), 554-559.
5. Miller, E. and W.E. Howden (eds.) (1981): *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society Press.
6. Myers, G.J. (1979): The Art of Software Testing, Wiley, New York.
7. Tratner, M. (1979): "A fundamental approach to debugging," *Software – Practice and Experience*, **9** (2), 97-100.
8. Voges, U. *et al* (1980): "SADAT: an automated testing tool," *IEEE Transactions on Software Engineering*, **6** (3), 286-290.
9. Wilcox, T.R. *et al* (1976): "The design and implementation of a table driven, interactive diagnostic programming system," *Commn. of ACM*, **19** (11), 609-616.
10. Woodward, M.R. and D. Hedley (1980): "Experience with path analysis and testing of programs," *IEEE Transactions on Software Engineering*, **6** (3), 278-286.

(d) Program instrumentation and Performance enhancement

1. Allen, F.E. and J. Cocke (1972): "A catalogue of optimizing transformations," in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs.
2. Arthur, J. and J. Ramanathan (1981): "Design of analyzers for selective program analysis," *IEEE Transactions on Software Engineering*, **7** (1), 39-51.
3. Dhamdhere, D.M., K.S. Shastry, C.V. Ravishankar and N.P.S. Bajwa, (1979): *IIT-FORT User Manual*, Publications Division, IIT Bombay.
4. Huang, J.C.(1978): "Program instrumentation and software testing," *Computer*, **11** (4), 25-33.
5. Ingalls, D. (1972): "The execution time profile as a programming tool," in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs.
6. Power, L.R.(1983): "Design and use of a program execution analyser," *IBM Systems Journal*, **22** (3), 271-294.

(e) Program generators

1. Martin, J.(1982): *Application Development Without Programmers*, Prentice-Hall, Englewood Cliffs.
2. Prywes, N.S., A. Pnueli and S. Shastry (1979): "Use of nonprocedural specification language and associated program generator in software development," *ACM TOPLAS*, **1** (2), 196-217.

(f) Structure editors and Programming environments

1. Allison, L. (1983): "Syntax directed program editing," *Software – Practice and Experience*, **13**, 453-465.
2. Barstow, D.R., H.E. Shrobe, and E. Sandewall (1983): *Interactive Programming Environments*, McGraw-Hill, New York.
3. Fischer, C.N. *et al* (1981): "An introduction to release I of editor Allan Poe," *Computer Science Technical Report 451*, University of Wisconsin.

4. Hunke, H. (1981): *Software Engineering Environments*, North Holland Publ. Co., Amsterdam.
5. Shapiro, E. *et al* (1981): "PASES – A programming environment for Pascal," *SIGPLAN Notices*, **16** (8), 50-57.
6. Teitelman, T. and T. Reps (1981): "The Cornell Program Synthesizer – a syntax directed programming environment," *Commn. of ACM*, **24** (9), 563-573.

(g) User Interfaces

1. Barth, P.S. (1986): "An OO approach to graphical interfaces," *ACM transactions on Graphics*, **5** (2), 142-172.
2. Hayes, F., N. Baran (1989): "A guide to GUI", *Byte*, July 1989, 250-257.
3. Larson (1991): *Interactive software: Tools for Building Interactive User Interfaces*, Prentice-Hall, Reading.
4. Mayhew (1992): *Principles and Guidelines in Software User Interface Design*, Prentice-Hall, Reading.
5. Olsen, D.R. (1992): *User Interface Management Systems*, Morgan Kaufmann, San Mateo, California.
6. Pfaff, G.E. (1985): *User Interface Management Systems*, Springer Verlag.
7. Schneiderman, B. (1983): "Direct manipulation : A step beyond programming languages," *Computer*, **16** (8), 57-69.
8. Schneiderman, B. (1993): *Designing the User Interface*, (2nd edition), Addison-Wesley, New York.