SYSTEMS PROGRAMMING

D M Dhamdhere

Tata McGraw-Hill

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

Systems Programming

First reprint 2011
RBZCRRBGRCQQZ

ISBN (13): 978-0-07-133311-5
ISBN (10): 0-07-133311-8

Vice President and Managing Director—McGraw-Hill Education, Asia Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Publishing Manager—SEM & Tech Ed.: *Shalini Jha*
Editorial Researcher: *Noaman Khan*
Sr Copy Editor: *Nimisha Kapoor*
Sr Production Manager: *Satinder S Baveja*

Marketing Manager—Higher Education: *Vijay S Jagannathan*
Sr Product Specialist—SEM & Tech Ed.: *John Mathews*

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

Typeset at Script Makers, 19, A1-B, DDA Market, Paschim Vihar, New Delhi 110 063 and text printed at Adarsh Printers, C-50-51, Mohan Park, Naveen Shahdara, Delhi – 110032

Cover Designer: Meenu Raghav

Cover Printer: A. B. Offset

The McGraw-Hill Companies

Contents

Preface

Easy development of new programs and applications, and their efficient operation on a computer are the primary concerns of computer users. However, both the concerns cannot be satisfied simultaneously. This situation has led to different kinds of schemes for program development which provide ease of program development to varying degrees and to many schemes for program execution which offer varying levels of execution efficiency. A user has to choose a scheme of each kind to obtain a suitable combination of ease of program development and efficiency of operation.

Programs that implement the schemes for program development and program execution are called *system programs* and the collection of system programs of a computer is called system software. A course on systems programming deals with the fundamental concepts and techniques used in the design and implementation of system programs, and their properties concerning speed of program development and efficiency of operation. Accordingly, a book on systems programming has to focus on numerous topics—support for quick and efficient development of programs, design of adaptive and extensible programs that can be easily modified to provide new functionalities, models for execution of programs written in programming languages, and interactions among user programs, the operating system, and the computer to achieve efficient and secure operation of the computer.

My previous book *Systems Programming and Operating Systems* covered the complete area of system software. It was used for three kinds of courses—systems programming, compilers, and operating systems. However, it was a large book that needed to get even larger to keep pace with developments in this field, so I decided to offer two separate books in this area. This book focuses primarily on systems programming. It can be used for a course on systems programming that contains a small module on operating systems and also for a course on compilers. It is expected that instructors and students of courses on operating systems would use my other book *Operating Systems—A Concept-Based Approach*.

General approach

Diversity of computer systems and system programs has been a major challenge in the writing of this text. I have used principles of *abstraction* and simple models of computer systems and system programs to present the key issues in design of system programs. This way, the text is free of dependence on specific computers, programming languages, or operating systems.

Pedagogical features

Chapter introduction The chapter introduction motivates the reader by describing the objectives of the chapter and the topics covered in it.

Figures and boxes Figures depict practical arrangements used to handle user computations and resources, stepwise operation of specific techniques, or comparisons of alternative techniques that project their strengths and weaknesses. Boxes are used to enclose key features of concepts or techniques being discussed. They also serve as overviews or summaries of specific topics.

Examples Examples demonstrate the key issues concerning concepts and techniques being discussed. Examples are typeset in a different style to set them apart from the main body of the text, so a reader can skip an example if she does not want the flow of ideas to be interrupted, especially while reading a chapter for the first time.

Algorithms Specific details of processing performed by system programs are presented in the form of algorithms. Algorithms are presented in an easy to understand pseudo-code form.

Case studies Case studies emphasize practical issues, arrangements and trade-offs in the design and implementation of specific schemes. Case studies are organized as separate sections in individual chapters.

Tests of concepts A set of objective and multiple choice questions are provided at the end of each chapter, so that the reader can test the grasp of concepts presented in the chapter.

Exercises Exercises are included at the end of each chapter. These include numerical problems based on material covered in the text, as well as challenging conceptual questions which test understanding and also provide deeper insights.

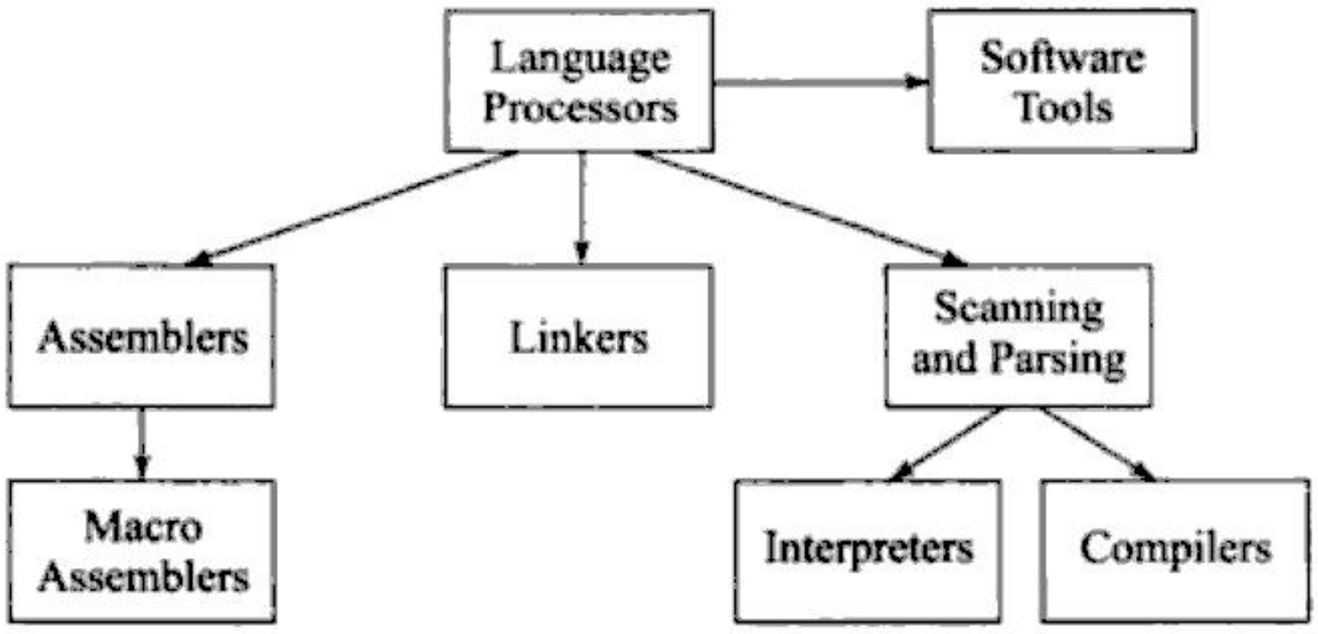
Chapter summary The summary included at the end of each chapter highlights the key topics covered in the chapter and their interrelationships.

Instructor resources A detailed solutions manual is provided.

Organization of the book

The first chapter describes goals of system software and discusses the characteristics that distinguish system programs from other kinds of programs. It also introduces the notion of *effective utilization* of a computer; it is a combination of user convenience, programmer productivity, and efficient and secure operation of a computer that suits a user's purposes. The rest of the chapter describes the broad spectrum of considerations that influence the design of system programs. These considerations range from user expectations such as high productivity during program development, ease of porting a program from one computer to another, ability to compose new programs by using existing programs as components, ability of software to adapt to its usage environment, and secure and efficient operation of programs.

The first chapter also develops two views of system software from the perspectives of ease of programming and efficient operation, respectively. The user-centric view is comprised of system programs that assist a user in fulfilling her computational needs. It includes system programs that transform a user program written in a programming language into a form that can be executed on a computer. The systemcentric view is comprised of programs that achieve effective utilization of a computer system by sharing its resources among user programs and ensuring non-interference during their execution. We call these two kinds of system programs *language processors* and *operating system programs*, respectively. The rest of the book is organized into two parts dealing with these two kinds of system programs, respectively.



Chapters of Part I

- **Part I: Language processors** A language processor is a system program that bridges the gap between how a user describes a computation in the form of a program, and how a computer executes a program. This gap is called the *specification gap*. The figure shows the interrelationship between chapters in this part.

  Chapter 2 describes how the specification gap influences ease of programming and execution efficiency of programs. It describes two classes of language processors called *program generators* and *translators*. It then describes fundamentals of the language processing activity and the organization of language processors. Remaining chapters of this part discuss details of specific kinds of language processors. Chapter 3 discusses the design of an *assembler*, which is the translator of a low-level machine-specific language called the assembly language. Chapter 4 discusses the *macro* facility provided in assembly languages, which enables a programmer to define new operations and data structures of her own choice to simplify design and coding of programs. Chapter 5 discusses *linkers* and *loaders*, which are system programs that merge the code of many programs so that they can be executed together and make the merged code ready for execution by the computer. Chapter 6 describes the techniques of *scanning* and *parsing* that are used by a language processor to analyse a program written in a programming language. Chapters 7 and 8 discuss *compilers* and *interpreters*, which are two models of execution of programs written in programming languages. Finally, Chapter 9 discusses *software tools*, which are programs that suitably interface a program with other programs to simplify development of new applications. Many of these tools use elements of language processing to achieve their goals.

- **Part II: Operating systems** The operating system controls operation of the computer and organizes execution of programs. Part II comprises five chapters. Chapter 10 describes the fundamentals of an operating system—how it controls operation of the computer, and how it organizes execution of programs. It contains an overview of those features of a computer's architecture that are relevant to the operating system and describes how an operating system functions. It then provides an overview of the concepts and techniques of operating systems used in the classical computing environments. Chapters 11–14 discuss concepts and techniques used in the four key functionalities of operating systems, namely, management of programs, management of memory, file systems, and security and protection in operating systems.

Using this book

Apart from an introduction to computing, this book does not assume the reader to possess any specific background. Hence it can be used by both students of systems programming and by working professionals.

Dhananjay Dhamdhere
April 2011

CHAPTER 1

Introduction

A modern computer has powerful capabilities such as a fast CPU, large memory, sophisticated input-output devices, and networking support; however, it has to be instructed through the machine language, which has strings of 0s and 1s as its instructions. A typical computer user does not wish to interact with the computer at this level. The *system software* is a collection of programs that bridge the gap between the level at which users wish to interact with the computer and the level at which the computer is capable of operating. It forms a software layer which acts as an intermediary between the user and the computer. It performs two functions: It translates the needs of the user into a form that the computer can understand so that the user's program can actually get executed on the computer. However, the computer has more resources than needed by a program, so many of its resources would remain idle while it is servicing one program. To avoid this problem, the software layer gives the idle resources to some other programs and interleaves execution of all these programs on the computer. This way, the computer can provide service to many users simultaneously.

Each program in the system software is called a *system program*. System programs perform various tasks such as editing a program, compiling it, and arranging for its execution. They also perform various tasks that a user is often unaware of, such as readying a program for execution by linking it with other programs and with functions from libraries, and protecting a program against interference from other programs and users. The term *systems programming* is used to describe the collection of techniques used in the design of system programs.

In this chapter we define the design goals of system programs and discuss the diverse functions performed by them. We then discuss the functions performed by different kinds of system programs. Details of their functioning and design are discussed in subsequent chapters.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

each computing environment, users desire a specific combination of convenience and efficient use. This is the notion of *effective utilization* of the computer system. For example, users in an interactive environment would favour convenience, e.g., fast response, to efficient use, whereas in a commercial data processing environment users would prefer efficiency to convenience because it would reduce the cost of computing. Hence an operating system simply chooses techniques that provide a matching combination of convenience and efficient use. We find a rich diversity in operating systems, and in system software in general, because effective utilization has a different flavour in each computing environment. We consider examples of this diversity in later sections of this chapter.

Interference with a user's activities may take the form of illegal use or modification of a user's programs or data, or denial of resources and services to a user. It could be caused by users of a computer system or by non-users. The system software must incorporate measures to prevent interference of all kinds and from all sources.

We discuss important aspects of the three fundamental goals of system software in the following section.

1.2.1 User Convenience

Table 1.1 lists many facets of user convenience. In the early days of computing, user convenience was synonymous with bare necessity—the mere ability to execute a program written in a higher level language was considered adequate. However, soon users were demanding better service, which in those days meant only fast response to a user command.

Table 1.1 Facets of user convenience

Facet	Examples
Fulfillment of necessity	Ability to execute programs, use the file system
Good Service	Speedy response to computational requests
User friendly interfaces	Easy-to-use commands, Graphical user interface (GUI)
New programming model	Concurrent programming
Web-oriented features	Means to set up web enabled servers
Evolution	Add new features, use new computer technologies

Other facets of user convenience evolved with the use of computers in new fields. Early operating systems had *command-line interfaces*, which required a user to type in a command and its parameters. Users needed substantial training to learn use of commands, which was acceptable because most users were scientists or computer professionals. However, simpler interfaces were needed to facilitate use of computers by new classes of users. Hence *graphical user interfaces* (GUIs) were designed. These interfaces used *icons* on a screen to represent programs and files and interpreted mouse clicks on the icons and associated menus as commands concerning

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

compare two system programs to decide which of them is 'better'. However, reality is more complex than this conception for several reasons.

An axis may not be homogeneous, so its calibration would be subjective and contentious. For example, how do we place the six facets of user convenience shown in Table 1.1 along the convenience axis in a non-controversial manner? Should we consider the ability to evolve more important than having web-oriented features? Similarly, given that a computer has several kinds of resources such as the CPU, memory and input-output devices, how do you assess efficient use of resources? Is one kind of resource more or less important than another kind? We could resolve this difficulty by developing a resource utilization function in which each kind of resource is a parameter. But if we did that we would have to decide how much weightage to assign to each of the resource kinds, and so on. The second reason is that the nature of the computing environment in which a system program is used decides how it rates along each of the axes. For example, the computational needs of a user decide what facet of user convenience and efficiency are relevant. Consequently, a system program cannot be rated uniquely along the three axes.

Due to these reasons, the question "Should system program A be preferred over system program B?" does not have a unique answer. In some situations A would be preferred while in some others B would be preferred. The purpose of studying system software is to know which of the two should be preferred in a specific situation. In the following sections we consider aspects which influence answers to this question. After a reader has gained a perspective for answering this question, she should try to answer the higher-level question "Under what situations should system program A be preferred over system program B?"

1.4.1 The Program Development and Production Environments

In a *program development environment*, users are engaged in developing programs to meet their computational needs. A user makes a trial run of her program to discover the presence of bugs, modifies the program to fix the bugs and makes further trial runs and so on until no new bugs can be discovered. In a *production environment*, the users merely execute already developed programs on one or more sets of data each to produce useful results; none of the programs face any modification.

A *compiler* and an *interpreter* are two system programs that can be used to execute programs in these environments. The compiler translates a program P written in a higher level programming language L into the machine language of a computer. Thus, it generates a machine language program that can be executed later. The interpreter does not generate a machine language program; instead it analyzes program P and directly carries out the computation described in it.

To understand comparative efficiency of a compiler and an interpreter, let us consider how they function: The compiler analyzes each statement in program P and generates a sequence of instructions in the machine language that would realize meaning of the statement when executed, by computing values, making decisions

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

this purpose as follows: When a client contacts the server, the client and the server coordinate their activities to download a program in the bytecode form in the client's computer, and the client initiates its interpretation by a Java virtual machine. This program obtains data from the server periodically. This scheme shifts most of the action to the client's side, thereby reducing the pressure on the server's resources. It is implemented simply by including a Java virtual machine in the client's web browser.

1.4.4 Treating Programs as Components

A new computational task to be accomplished may have known subtasks for which programs have been already developed. A program for performing the new task can be developed quickly and cheaply if the existing programs can be treated as its components. This requirement points to a facility that permits programs to be *glued* together to form larger software systems. A user interface is a prime example of such a facility because it permits ready programs—either system programs provided in the system software of a computer or those developed by users—to be invoked to accomplish larger tasks. A language designed for gluing together existing program components is called a *scripting language*. Note that the scripting language merely binds together those programs that actually perform the computational task. Hence efficiency of the scripting language is not important.

The scripting languages provided in early mainframe computer systems were called *job control languages* (JCL). In such computer systems, a job consisted of a sequence of programs. The JCL allowed a programmer to indicate which programs constituted a job and implemented their execution in the specified sequence. The shell of Unix and graphical user interfaces (GUIs) of the Apple, Windows and Linux operating systems are current-day descendants of the JCL. Example 1.1 contains an illustration of the Unix shell. A scripting language used in this manner is also called the *command language* of an operating system.

Example 1.1 (Gluing of programs using the Unix shell) The following command in the Unix shell

```
cat alpha | sort | uniq | wc -l
```

glues together four programs to accomplish counting of unique names in file `alpha`. These programs are—`cat`, `sort`, `uniq` and `wc`. '`|`' is the symbol for a Unix pipe, which sends the output of one program as input to another program. Thus, the output of `cat` is given to `sort` as its input, the output of `sort` is given to `uniq` and the output of `uniq` is given to `wc`. Program `cat` reads the names from file `alpha` and writes each name into its standard output file. The output of `cat` is the input to `sort`, so `sort` reads these names, sorts them in alphabetical order, and writes them into its output file. `uniq` removes duplicate names appearing in its input and outputs the unique names. `wc` counts and reports the number of names in its input. (Note that `-l` is a parameter passed to the `wc` program asking it to count the number of lines in its input because `uniq` puts each name in a line by itself.)

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

100 real elements. The compiler makes use of this type information and generates appropriate machine language instructions for accessing these variables. Some elements of array `alpha` may not be used during an execution of the program; however, the compiler would have allocated memory to them. Consequently, some of the allocated memory may never be used.

```
i, m :  integer;
j :  real;
m := 50;
alpha [1 .. 100]:  real;
if m < 50 then j := m - 10.5;
   else j := m - 56;
i := j + 6;
alpha[i] := ...;
end;
```

A program in language L_1

```
m := 50;
alpha [1 .. m];
if m < 50 then j := m - 10.5;
   else j := m - 56;
i := (integer) j + 6;
alpha[i] := ...;
end;
```

A program in language L_2

Figure 1.4 Static and dynamic specification in programs

In language L_2, types and dimensions of variables are either specified dynamically or inferred from the context of their use. Variable `j` would be of type real if `m` is < 50 because value of the expression `m - 10.5` is assigned to it, but it would be of type integer if `m` $\not< 50$. However, irrespective of `j`'s type `i` would be of type integer because of the specification "`(integer)`" appearing in the right-hand side of the assignment to `i`. The size of array `alpha` is determined by the value of `m`; presumably, this manner of specifying the size avoids wastage of memory due to unused array elements. Because the type of a variable depends on the value assigned to it, a variable may have different types in different parts of a program. Such a program cannot be compiled. It would have to be interpreted, which would slow down its execution (see Section 1.4.1).

Flexibility is the capability to broaden the choice in a specification or decision according to the needs of a user. In the world of programming languages, flexibility is provided through *user defined data types*. A user can define her own data type for use in a program, say type T_k, by specifying the following:

- The values that a variable of type T_k may assume,
- The operations that can be performed on variables and values of type T_k.

Now the set of types available in the program is comprised of the built-in types of the language, such as types `real` and `integer` in Figure 1.4, and the user defined data types such as type T_k.

Adaptive software is one that adjusts its own features and behavior according to its environment. As we shall see shortly, it uses both dynamic features and flexibility. The *plug-and-play* capability of an operating system is an example of adaptive behavior. It allows new devices to be connected and used during the operation of a computer. The operating system implements this capability as follows: The computer has a feature through which it alerts the operating system when a device is

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Some of the functions performed by system programs included in the system-centric view are described in the following.

Resource sharing is implemented through the following two techniques: *resource partitioning* divides the resources among programs such that each program has some resources allocated for its exclusive use, whereas *resource scheduling* selects one of the many programs that wish to use a scarce resource and allocates the resource to it for some duration of time. Memory, storage devices such as disks, and input-output devices such as keyboards and screens are handled by means of resource partitioning, whereas the CPU is subject to scheduling.

Scheduling of the CPU results in interleaved execution of programs. An operating system uses a scheduling policy that provides an appropriate combination of user convenience and efficient use of resources in a computing environment. A multi-user application software needs to service many users' requests simultaneously. An operating system provides *multithreading* support for this purpose.

1.6 OVERVIEW OF THE BOOK

As discussed in earlier sections, the *system software* of a computer is a collection of system programs and *systems programming* is the collection of techniques used in the design of system programs.

A dominant theme in system programming is the trade-off between *static* and *dynamic* actions. A static action is one that is performed before execution of a program is started. A dynamic action is performed during execution of a program. The action incurs an overhead during the program's execution; however, it provides flexibility because it can use information that is determined during a program's execution. We have seen instances of both overhead and flexibility of dynamic decisions when we discussed different kinds of system programs in Section 1.4. So system programming involves a careful analysis of the benefits of static and dynamic decision making and actions. It results in design of appropriate data structures for use in a program and schemes for use in language processors and operating systems.

In Section 1.5, we discussed how system programs can be grouped into two views that we called the *user-centric* and *system-centric* views of system software. The user-centric view consists of language processors that are used for developing new programs, and system programs that assist in the use of existing programs. It also contains operating system programs that help in execution of user programs; however, these programs and their functions are not as visible to the users. The system-centric view consists of programs that focus on efficient use of a computer system and non-interference with execution of programs and use of resources. It contains programs of the operating system that work harmoniously to achieve a common goal. Hence the special techniques used in their design are called *operating system techniques*.

Accordingly, chapters of this book are organized into two parts. The first part discusses the system programming techniques used in language processors and in

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

this decision. Such software adjusts its own features and behavior according to its environment. For example, it can switch between the use of a compiler and an interpreter depending on whether it finds that a program is being executed for production purposes or debugging purposes.

Some system programs are used to provide benefits that are not restricted to specific computing environments. A *virtual machine* is implemented by a system program to provide easy *portability* of programs. Virtual machines are employed to provide a capability to download programs from the Internet and execute them on any computer. *Scripting languages* permit new programs to be composed from existing programs and also provide a method of quickly implementing certain kinds of computational tasks.

System software can be viewed from two perspectives. The user-centric view focuses on language processors and scripting languages, while the system-centric view focuses on programs that provide effective utilization of a computer, i.e., programs in the operating system. We study language processors and scripting languages in Part 1 of the book and operating systems in Part 2 of the book.

TEST YOUR CONCEPTS

1. Classify each of the following statements as true or false:
   (a) An abstract view hides unimportant details.
   (b) User convenience and efficient use of a computer system are completely independent of one another.
   (c) Efficient use of a computer system is of primary importance in a program development environment.
   (d) Use of a virtual machine enhances portability of software.
   (e) A scripting language can glue together existing programs.
   (f) User convenience is of primary importance in the embedded systems environment.
   (g) User defined data types provide flexibility.
   (h) A just-in-time compiler uses adaptive techniques.
   (i) The system-centric view of system software contains language processors.

BIBLIOGRAPHY

The XEN virtual machine product is described in Barham et al. (2003). Rosenblum and Garfinkel (2005) discuss trends in the design of virtual machine monitors. The May 2005 issue of *IEEE Computer* is a special issue on virtualization technologies.

Welch (2003), Schwartz et al. (2008), and Beazley (2009) are books devoted to the scripting languages Tcl/Tk, Perl, and Python, respectively. Aycock (2003) discusses just-in-time compilers.

1. Aycock, J. (2003): A brief history of just-in-time, *ACM Computing Surveys*, 35 (2), 97–113.

2. Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003): "XEN and the art of virtualization," *ACM Symposium on Operating System Principles*, 164–177.
3. Beazley, D. M. (2009): *Python Essential Reference*, fourth edition, Addison-Wesley Professional, Reading, MA.
4. Rosenblum, M. and T. Garfinkel (2005): "Virtual machine monitors: current technology and future trends," *IEEE Computer*, 38 (5), 39–47.
5. Schwartz, R., T. Phoenix, and B. D. Foy (2008): *Learning Perl*, fifth edition, O'Reilly Media, Sebastopol.
6. Smith, J. E. and R. Nair (2005): *The Architecture of Virtual Machines*, *IEEE Computer*, 38 (5), 32–38.
7. Welch, B. B. (2003): *Practical Programming in Tcl and Tk*, Prentice-Hall, N.J.

PART I

LANGUAGE PROCESSORS

CHAPTER 2

Overview of Language Processors

A user would like an arrangement in which she could describe a computation in some convenient manner and the system software of the computer system would implement the computation using the computer.

A *language processor* is a system program that bridges the gap between how a user describes a computation—we call it a *specification* of the computation—and how a computer executes a program. The ease of specification depends on the language in which the specification is written. A *problem oriented programming language* lets a user specify a computation using data and operations that are meaningful in the application area of the computation. A *procedure oriented programming language* provides some standard methods of creating data and performing operations and lets the user describe the intended computation by using them. Of the two, use of a problem oriented language provides more user convenience.

Two kinds of language processors are used to implement a user's computation. A *program generator* converts the specification written by the user into a program in a procedure oriented language, whereas a compiler or interpreter helps in implementing a program written in a programming language. A *compiler* translates a program into the *target language*, which is either the machine language of a computer or a language that is close to it. An *interpreter* analyzes a program and itself performs the computation described in it with the help of the computer. The compiler and interpreter suit the needs of different kinds of programming environments.

We discuss the fundamentals of a language processor in this chapter—how it analyzes a program input to it and how it synthesizes a program in the target language. It uses a data structure called a *symbol table* to store attributes of symbols used in a program. The design of the symbol table is a crucial decision because a language processor accesses it a large number of times. We describe the principle of *time–space trade-off* used in its design and discuss many symbol table organizations.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

of converting specifications across any pair of domains is simpler, so it can be performed more reliably.

- A system program can be used to perform conversion of specifications across domains. Now the conversion is faster, cheaper and more reliable because it is not performed manually. Of course, correctness of the system program that performs the conversion would have to be ensured, but that is a one-time effort.

Use of a programming language combines both these methods as follows: The programming language domain is introduced as an intermediate domain, so the semantic gap between the application and execution domains is split into two smaller semantic gaps—the specification gap and the execution gap (see Figure 2.2). Only the specification gap has to be bridged by the software designer, whereas the execution gap is bridged by the language processor such as a compiler or an interpreter. The language processor also provides a capability to detect and indicate errors in its input, which helps in improving reliability of software.

We assume that each domain has a specification language. A specification written in a specification language is a *program* in the specification language. The specification language of the programming language domain is the programming language itself. The specification language of the execution domain is the machine language of the computer system. We use 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 semantic gap between the semantics of programs that perform the same task but are written in different programming languages.

We restrict 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.

2.1.1 Kinds of Language Processors

Definition 2.1 (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. As mentioned earlier, a semantic gap is bridged by converting a specification in one domain into a specification in another domain. During this conversion, the language processor points out errors in the input specification and aborts the conversion if errors are present. This capability of a language processor, which we call the *diagnostic capability*, contributes to reliability of a program written in a programming language by ensuring that it would reach execution only if it is free of specification errors. (We shall discuss the diagnostic capability of language processors in Chapters 6 and 7.)

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- *Program generation activity:* This activity generates a program from its specification. The language in which the specification of the program is written is close to the specification language of an application domain and the target language is typically a procedure oriented programming language. Thus, program generation bridges the specification gap.
- *Program execution activity:* This activity aims at bridging the execution gap by organizing execution of a program written in a programming language on a computer system. The programming language could be a procedure oriented language or a problem oriented language.

2.2.1 Program Generation

Figure 2.6(a) depicts the program generation activity. The *program generator* is a system program which accepts the specification of a program in some specification language and generates a program in the target language that fulfills the specification. Use of the program generator introduces a new domain called the *program generator domain* between the application and programming language domains (see Figure 2.6(b)). The specification gap faced by the designer of an application program 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 programming language domain; the gap is insignificant if the specification language is a problem oriented language. The execution gap between the target language domain and the execution domain is bridged by a compiler or interpreter for the programming language.

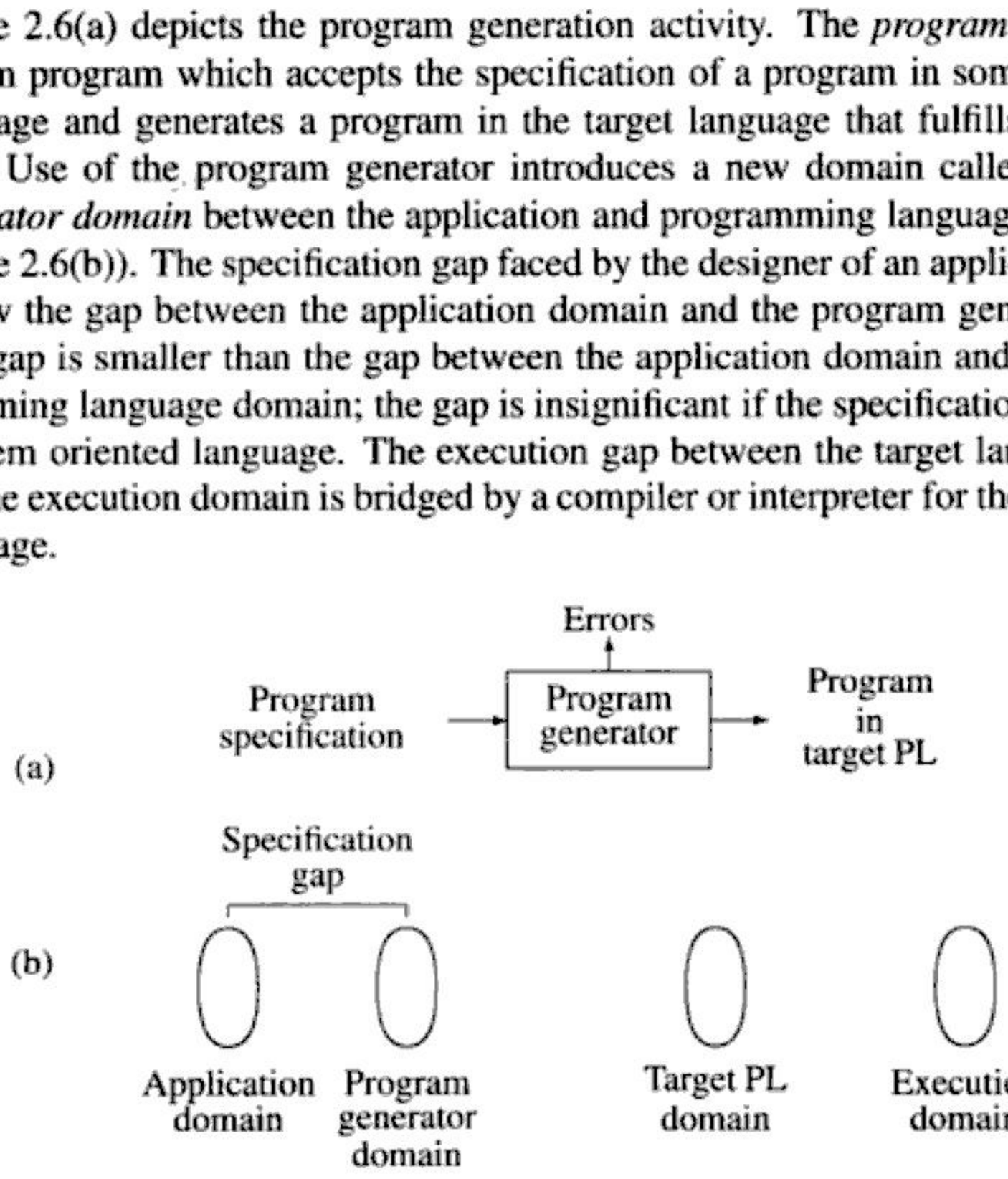


Figure 2.6 Program generation: (a) Schematic, (b) the program generator domain

If the program generator domain is close to the application domain, the specification gap is small. Consequently, the task of writing the specification is simple, which increases the reliability of the generated program. The harder task of bridging

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- If a source program is modified, the modified program must be translated before it can be executed. We call it *retranslation* following a modification.

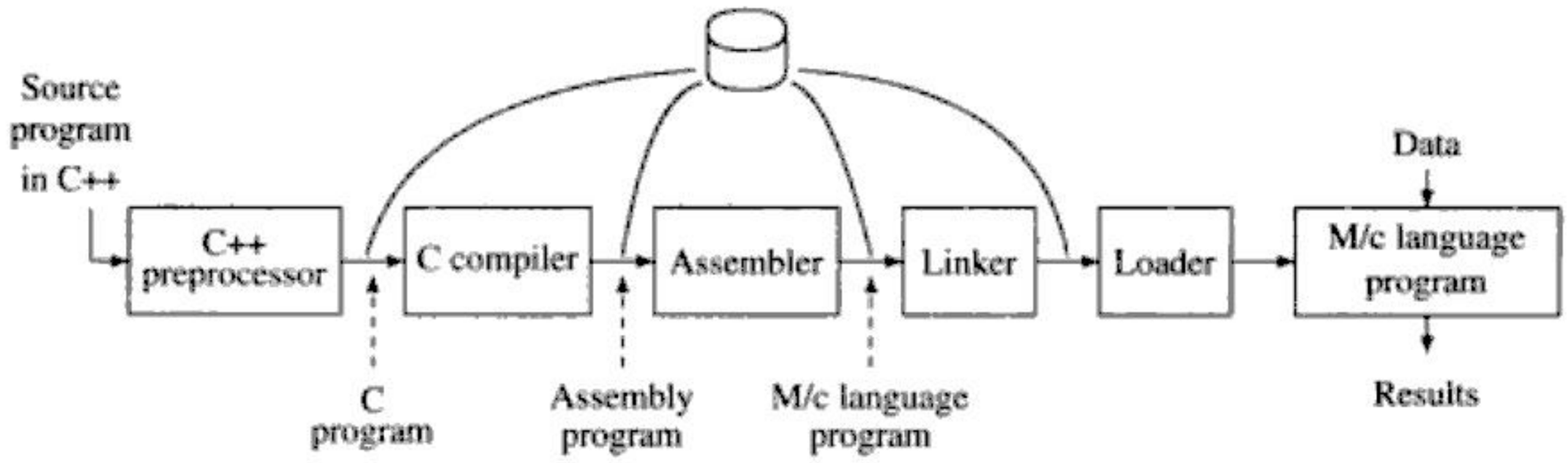


Figure 2.10 A practical arrangement of language processors

Practical arrangements for program execution differ from that depicted in Figure 2.9 for providing higher effectiveness and flexibility. Figure 2.10 shows a practical arrangement for executing C++ programs which uses a number of language processors. The output of each of the language processors can be saved on disk and used repeatedly. Important features of this arrangement are described below.

- *Preprocessors:* One or more preprocessors may be used along with a translator for a programming language to provide a superset of the programming language's features. Recall from Example 2.1 that use of the C++ preprocessor along with a C compiler enables use of the C++ language without having to develop a C++ compiler. In Chapter 5 we shall discuss use of a *macro preprocessor* for providing a superset of the features of an assembly language.
- *Using a sequence of translators:* Translation from a programming language to a machine language may be achieved by using two or more translators. The translator used in the first step produces a target program that is not in the machine language. This target program is input to another translator, and so on, until we obtain a target program that is in the machine language. Each of the translators is less complex than a translator for the programming language that would have directly produced a machine language program. This arrangement also has another benefit that is described later.
- *Linking and loading:* A target program obtained by translating a program written in a programming language requires the help of some other programs during its execution, e.g., programs that perform input-output or standard mathematical functions. The *linker* is a system program that puts all these programs together so that they can execute meaningfully. The *loader* is a program that loads a ready-to-run program from a file into the computer's memory for execution.

If one of the intermediate target programs is in a standard programming language, say, language PL_i, the arrangement using a sequence of translators can be

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

declaration statements. Some language processors such as assemblers can perform this task equally well in the analysis phase.

- If the source statement is an *imperative statement*, that is, a statement indicating that some actions should be performed, decide how the actions should be performed in the target language and generate corresponding statements.

We refer to these tasks as *memory allocation* and *code generation*, respectively. Example 2.4 illustrates synthesis of the target program.

Example 2.4 (Synthesis of target program) A language processor generates the following target program for the source statement of Example 2.3.

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

This program is in the assembly language of a computer, which we call the *target machine* of the language processor. The `DW` statements in the program reserve words in memory for holding the data, the `DC` statement reserves a word in memory and stores the constant 100 in it, and the statements `MOVER` and `MOVEM` move a value from a memory location to a CPU register and vice versa, respectively. Needless to say, both memory allocation and code generation are influenced by the target machine's architecture.

2.3.1 Multi-Pass Organization of Language Processors

The schematic of Figure 2.12 and Examples 2.3 and 2.4 may give the impression that language processing can be performed on a statement-by-statement basis—that is, analysis of a statement in the source program can be immediately followed by synthesis of target statements that are equivalent to it. However, statement-by-statement processing of the source program may not be feasible due to the following two reasons:

1. A source program may contain *forward references*.
2. A language processor that performs statement-by-statement processing of a source program may require more memory than is available for its operation.

We discuss these issues below.

Definition 2.3 (Forward reference) *A* forward reference *of a program entity is a reference to the entity in some statement of the program that occurs before the statement containing the definition or declaration of the entity.*

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

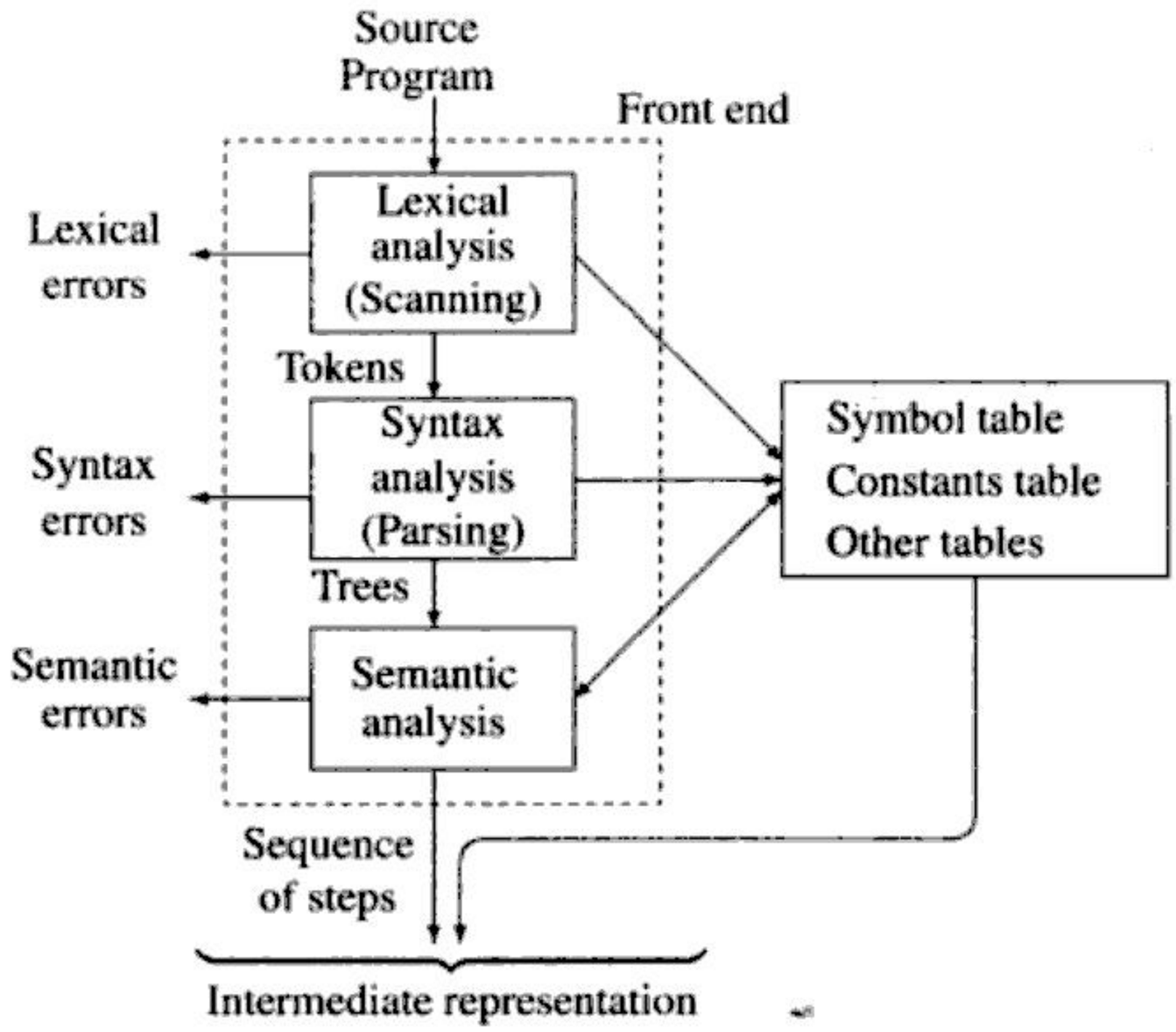


Figure 2.15 Front end of the toy compiler

Figure 2.15 shows how tables and intermediate codes are generated and used in the front end. Each analysis in the front end represents the 'content' of a source statement in a suitable form. Subsequent analysis uses this information for its own purposes and either adds information to this representation of content or constructs its own representation. Note that *scanning* and *parsing* are technical terms for lexical analysis and syntax analysis, respectively.

Lexical analysis (Scanning)

Lexical analysis considers the source program as a string of characters. It identifies smaller strings that are lexical units, classifies them into different lexical classes, e.g., operators, identifiers, or constants; and enters them into the relevant tables. The classification is based on the specification of the source language. For example, an integer constant is a string of digits with an optional sign, an identifier is a string of letters, digits and special symbols whose first character is a letter, whereas a *reserved word* is a string that has a fixed meaning in the language.

Lexical analysis builds a separate table for each lexical class in which it stores information concerning the lexical units of that class, e.g., a table of identifiers. It builds an intermediate code that is a sequence of intermediate code units (IC units) called *tokens*, where a token is a descriptor for a lexical unit. A token contains two fields—*lexical class* and *number in class*. The *number in class* is a unique number within the lexical class that is assigned to the lexical unit—we simply use the entry number of a lexical unit in the relevant table as the number in class. For example, the

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

2.3.2.2 The Back End

The back end performs memory allocation and code generation. Figure 2.20 shows its schematic.

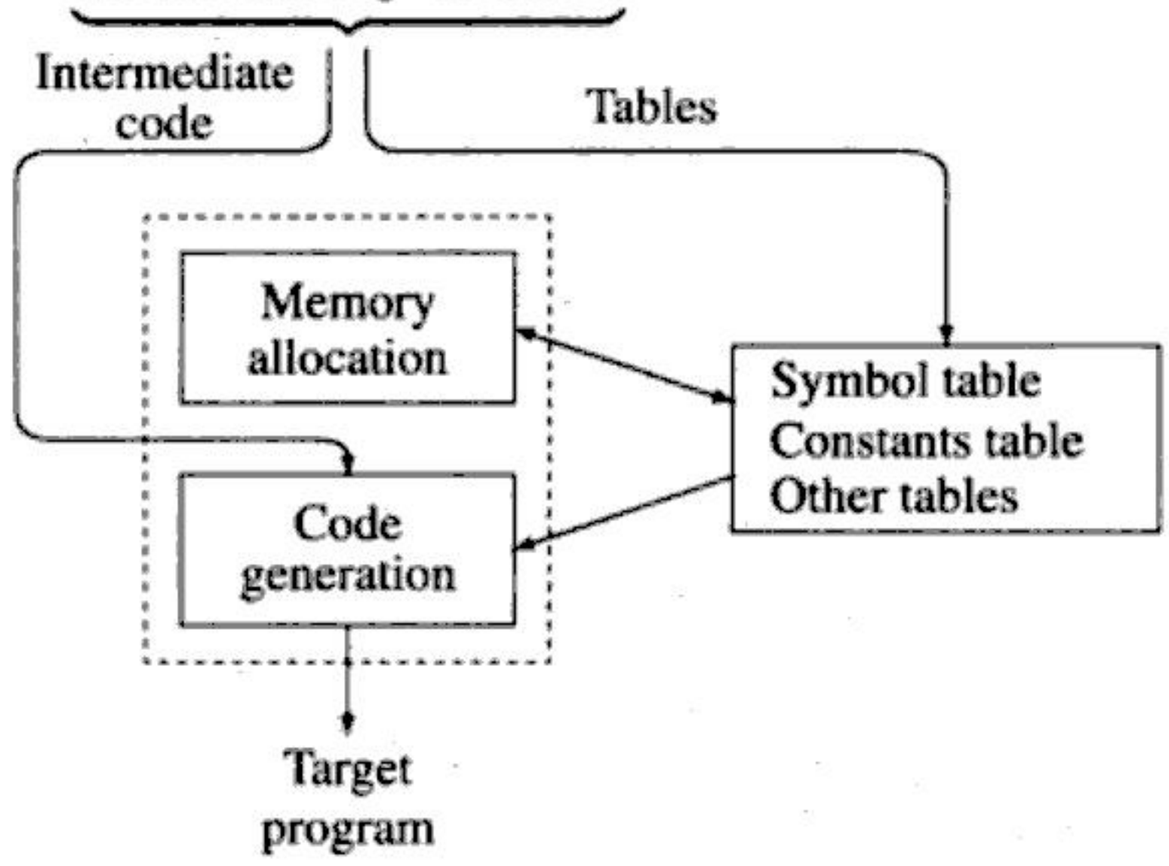


Figure 2.20 Back end of the toy compiler

Memory allocation

The back end computes the memory requirement of a variable from its type, length and dimensionality information found in the symbol table, and allocates memory to it. The address of the allocated memory area is entered in the symbol table. Note that certain decisions have to precede memory allocation. The temporary results `i*` and `temp` of Example 2.9 are both computed and used in the assignment statement `a = b+i`. Hence they could be held in registers of the CPU if some registers are available; otherwise, they would have to be stored in memory so memory should be allocated to them.

Example 2.10 (Memory allocation) After memory allocation, the relevant information in the symbol table looks as shown in Figure 2.21. It is assumed that each variable requires only one memory location, and that `temp` and `i*` are not stored in memory.

Code generation

Complex decisions are involved in generating good quality target code. Two key decisions are as follows:

1. What instructions should be used for each of the actions in the intermediate code?
2. What CPU registers should be used for evaluating expressions?

Example 2.11 illustrates the target code resulting from these decisions.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example 2.13 (Fixed-length and variable-length entries in the symbol table) Figure 2.23 shows two entry formats for the symbol table according to the attributes listed in Table 2.1. Part (a) shows the fixed-length entry format. Since total 8 attributes are specified, the entry has 8 attribute fields. When *class* = label, all attribute fields except the field for storing the statement number are redundant. Part (b) shows the variable-length entry format for a symbol that is used as a label. It has only one attribute field because symbols of the label class have only one attribute.

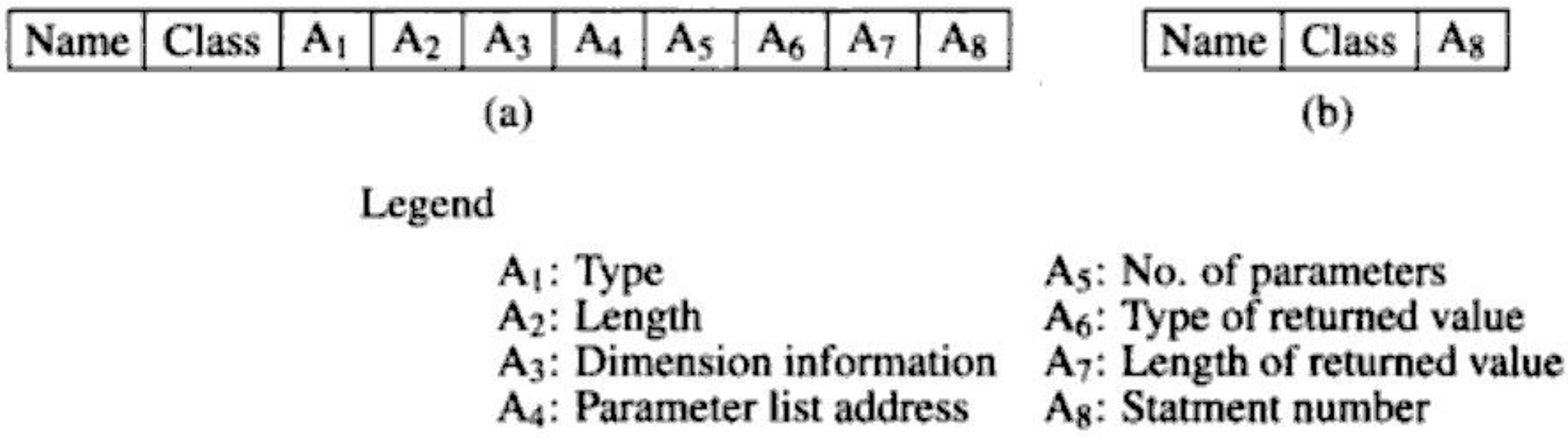


Figure 2.23 (a) Fixed-length entry, (b) Variable-length entry for a label

When the fixed-length entry format is used, all entries in the symbol table have an identical format. It enables the use of homogeneous linear data structures like arrays for a symbol table. As we shall see later in the section, use of a linear data structure enables the use of an efficient search procedure. However, this organization makes inefficient use of memory since many entries may contain redundant fields—the entry for a label contains 7 redundant fields in Example 2.13.

Use of the variable-length entry format leads to a compact organization in which memory wastage does not occur. However, the search method would have to know the length of an entry, so each entry would have to include a *length* field. We will depict the format of such an entry as

The hybrid entry format is used as a compromise between the fixed and variable-length entry formats to combine the search efficiency of the fixed-length entry format with the memory efficiency of the variable-length 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 a structure that facilitates efficient search, e.g., a linear data structure. Since the fixed part of an entry contains a pointer to its variable part, the variable part does not need to be located through a search. Hence it can be put into a linear or nonlinear data structure. We will depict the format of the hybrid entry as

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

organization faced due to physical deletion of entries. Hence the binary search organization is not a good choice for a symbol table. However, it is a suitable organization for a table that contains a fixed set of symbols, e.g., the table of reserved words in a programming language.

Hash table organization

In the hash table organization the guess in Algorithm 2.1 about the entry occupied by a symbol s depends on the symbol itself, that is, e is a function of s. Three possibilities arise when the e^{th} entry is probed—the entry may be occupied by s, the entry may be occupied by some other symbol, or the entry may be unoccupied. The situation where the probed entry is occupied by some other symbol, that is, $s \neq s_e$, is called a *collision*. Following a collision, the search continues with a new guess. If the probed entry is unoccupied, symbol s does not exist in any entry of the table, so we exit with the flag set to *failure*. If the symbol is to be added to the table, this probed entry should be used for it.

Algorithm 2.3 (Hash table management)

1. $e := h(s)$;
2. If $s = s_e$, exit with the flag set to *success* and the entry number e. If entry e is unoccupied, exit with the flag set to *failure*.
3. Repeat Steps 1 and 2 with different functions h', h'', etc., until we either locate the entry for s or find an unoccupied entry.

The function h used in Algorithm 2.3 is called a *hashing function*. In the following, we discuss what properties a hashing function should have so that the number of probes required to locate a symbol's entry is small. We use the following notation in our discussion:

n	:	Number of entries in the table
f	:	Number of occupied entries in the table
ρ	:	*Occupation density* in the table, which is 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, that is, the space formed by the entries 1 ... n
K	:	*Key space of a programming language*, that is, the space formed by enumerating all valid symbols possible according to the specification of the source language. We will denote it as 1 ... k, where k is the number of valid symbols.
K_p	:	*Key space of a source program*, that is, the space formed by enumerating all symbols used in a program. We will denote it as 1 ... k_p. k_p^m is the largest value k_p may have.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

this performance has been 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, i.e., the largest number of symbols a program is expected to contain. If a program contains fewer symbols, ρ would be smaller and 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 performance is independent of the size of the table; it is determined only by the value of ρ.

Table 2.2 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 for this purpose. They focus on dispersing the colliding entries to reduce the average cluster size. For example, the sequential step rehash scheme uses the recurrence relation $h_{i+1}(s) = [h_i(s) + i - 1] \bmod n + 1$, which would put symbols `a`, `b`, `c`, and `d` in entries 5, 6, 8, and 7 in Example 2.14, whereby `d` would suffer only one collision instead of 2.

Overflow chaining

The overflow chaining organization uses two tables called the *primary table* and the *overflow table*, which have identical entry formats, and a single hashing function h. To add a symbol, a probe is made in the primary table using h. If no collision occurs, the symbol is added to the probed entry; otherwise, it is accommodated in the overflow table. A search is conducted in an analogous manner: If a symbol is not found in the primary table, its search has to be continued in the overflow table. To avoid probing all entries in the overflow table during a search, entries of symbols that encountered a collision in the same entry of the primary table are chained together using pointers. This way, the overflow table would contain many chains of entries and only entries in a specific chain have to be searched to locate a symbol. To facilitate this arrangement, a pointer field is added in each entry in the primary and overflow tables. Accordingly, each entry has the following form:

Symbol	*Other info*	*Pointer*

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

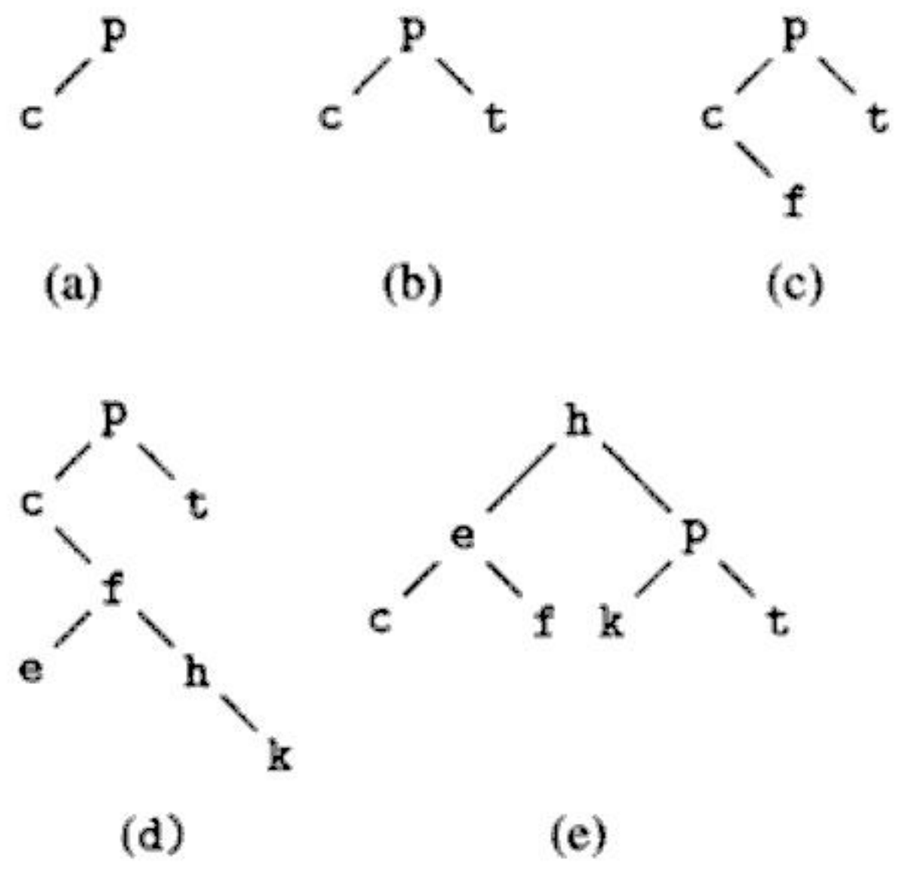


Figure 2.27 Tree structured table organization

only symbol in the subtree that would be accessed using the right pointer in the entry of p. All other symbols are stored in the subtree that would be accessed using the left pointer in the entry of p. Consequently a *locate* operation on t would require fewer probes than a locate operation on, say, k. Part (e) shows a balanced binary tree with the same symbols which would provide better search performance if all entries in the table are accessed with the same probability. However, reorganizing the tree of Part (d) to the form shown in Part (e) would itself incur a processing cost. More sophisticated data structures such as the B+ tree may be used to limit this overhead.

Search along a secondary dimension

In some situations it is useful to support a search along a secondary dimension within a symbol table. Since the search structure cannot be linear in the secondary dimension, a linked list representation is used for the secondary search. Such search structures are useful for handling fields of records in Pascal, PL/1, and Cobol and structures of C, members of objects in Java and parameters of functions in any source language. Example 2.18 illustrates a generalization of the search structure which allows searches along several dimensions; it is known as a *multi-list structure*.

Example 2.18 (Multi-list-structured symbol table) The following fragment of a C program declares variable `personal_info` as a structure with `name`, `sex` and `id` as its members.

```
struct{
        character name[10];
        character sex;
        integer id;
      } personal_info;
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

BIBLIOGRAPHY

Czarnecki and Eisenecker (2000) survey the methods and tools of generative programming. Kiczales et al. (1997) and Filman et al. (2004) discuss aspect-oriented programming. The October 2001 issue of *Communications of the ACM* is devoted to aspect-oriented programming.

Table management crucially determines the speed of language processing. Most books on compilers include a good coverage of table management techniques. Bell (1970) and Ackerman (1974) deal with rehashing schemes. Horowitz, Sahni and Anderson (1992) and Tremblay and Sorenson (1985) are good sources on algorithms for table management. Jones and Lins (1996) describe garbage collection techniques.

1. Ackermann, A. F. (1974): "Quadratic search for hash tables of size p^n," *Communications of the ACM*, 17 (3), 164–165.
2. Bell, J. R. (1970): "The quadratic quotient method," *Communications of the ACM*, 13 (2), 107–109.
3. Czarnecki, K. and U. Eisenecker (2000): *Generative Programming—Methods, Tools and Applications*, Addison-Wesley, Boston.
4. Filman, R. E., T. Elrad, S. Clarke, and M. Aksit (2004): *Aspect-Oriented Software Development*," Addison-Wesley Professional.
5. Horowitz, E., S. Sahni, and S. Anderson-Freed (1992): *Fundamentals of Data Structures in C*, W. H. Freeman.
6. Jones, R. and R. Lins (1996): *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley and Sons.
7. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin (1997): "*Aspect-oriented programming*," Proceedings of the European Conference on Object-Oriented Programming, Vol. 1241, 220–242.
8. Knuth, D. (1973): *The Art of Computer Programming, Vol. III – Sorting and Searching*, Addison-Wesley, Reading.
9. Tremblay, J. P., and P. G. Sorenson (1985): *An Introduction to Data Structures with Applications*, second edition, McGraw-Hill.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

can be tested by a Branch on Condition (BC) instruction. The assembly statement corresponding to it has the format

BC *<condition code specification>*, *<memory address>*

It transfers control to the memory word with the address *<memory address>* if the current value of condition code matches *<condition code specification>*. For simplicity, we assume *<condition code specification>* to be a character string with obvious meaning, e.g., the strings GT and EQ indicate whether the result is > 0 and $= 0$, respectively. A BC statement with the condition code specification ANY implies unconditional transfer of control.

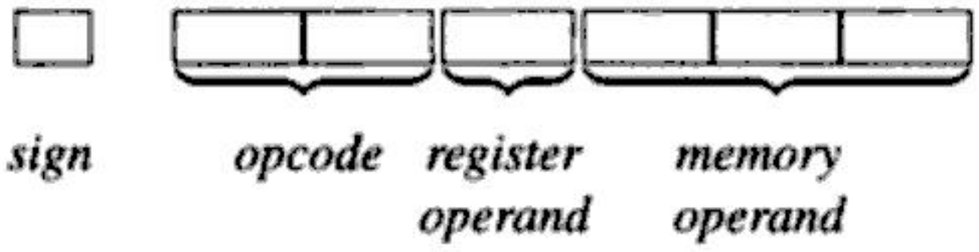


Figure 3.2 Instruction format

Figure 3.2 shows the format of machine instructions. 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.

3.1.1 Assembly Language Statements

The assembly language has three kinds of statements—imperative statements, declaration statements, and assembler directives.

Example 3.1 (A sample assembly language program) Figure 3.3 shows an assembly language program for computing $N!$ and the corresponding machine language program generated by an assembler. For simplicity, we show all addresses and constants in a machine language program in decimal rather than in the binary, octal or hexadecimal notation typically used in a computer. The DS and DC statements are *declaration statements*. The START and END statements are *directives* to the assembler. The START directive provides the address that is to be given to the first memory word in the machine language program.

Imperative statements

An imperative statement indicates an action to be performed during the execution of the program, e.g., an arithmetic operation. Each imperative statement translates into one machine instruction.

Declaration statements

The syntax of declaration statements is as follows:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

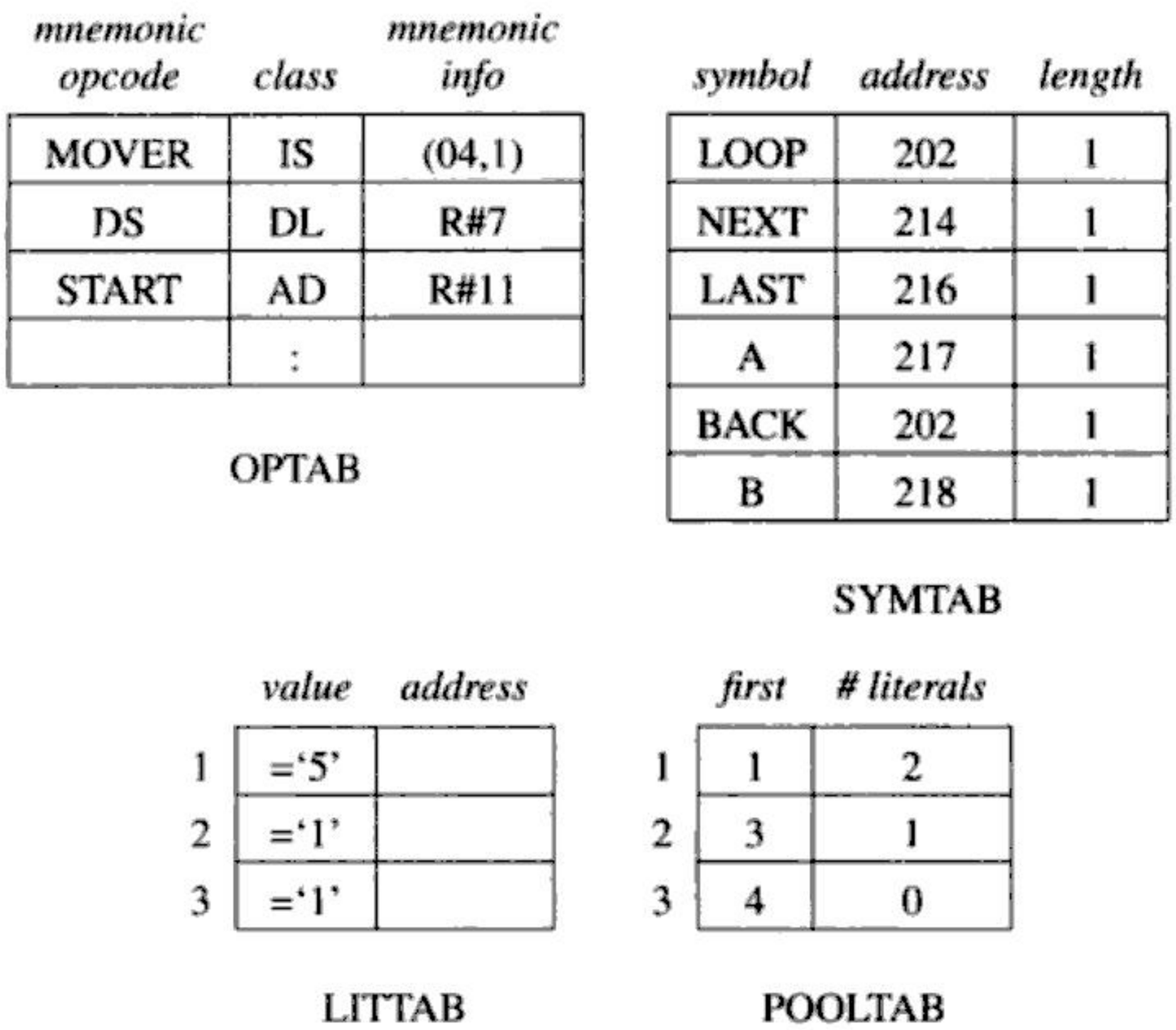


Figure 3.9 Data structures of assembler Pass I

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 location counter. The length is also entered in the SYMTAB entry of the symbol (if any) defined in 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 processes the operand field of the statement to determine the amount of memory required by this statement, which we call its em size. It returns the size and a code that is to be put into the intermediate code to describe the declaration or assembler directive statement. The routine for an EQU statement enters the address represented by *<address specification>* in the SYMTAB entry of the symbol appearing in the label field of the statement and enters 1 in its length field. The routines for other assembler directives perform appropriate processing, possibly affecting the address contained in the location counter.

The assembler uses the LITTAB and POOLTAB as follows: 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 address in the location counter and the address in the location counter is appropriately incremented. Example 3.7 illustrates handling of literals in the first

pass.

Example 3.7 (Handling of literal pools) The assembler allocates memory to the literals used in the program of Figure 3.8(a) in two steps. At start, it enters 1 in the first entry of POOLTAB to indicate that the first literal of the first literal pool occupies the first entry of LITTAB. The Literals ='5' and ='1' used in Statements 2 and 6, respectively, are entered in the first two entries of the LITTAB. At the LTORG statement, these two literals will be allocated the addresses 211 and 212 and the entry number of the first free entry in LITTAB, which is 3, will be entered in the second entry of POOLTAB. The literal ='1' used in Statement 15 will be entered in the third entry of LITTAB. On encountering the END statement, this literal will be allocated the address 219.

The assembler implements Pass I by using Algorithm 3.1. It uses the following data structures:

OPTAB, SYMTAB, LITTAB and POOLTAB
LC : Location counter
littab_ptr : Points to an entry in LITTAB
pooltab_ptr : Points to an entry in POOLTAB

Details of the intermediate code generated by Pass I are discussed in the next section.

Algorithm 3.1 (Pass I of a two-pass assembler)

1. LC := 0; (This is the default value)
   littab_ptr := 1;
   pooltab_ptr := 1;
   POOLTAB [1].*first* := 1; POOLTAB [1].*# literals* := 0;
2. While the next statement is not an END statement
   (a) If a symbol is present in the label field then
   this_label := symbol in the label field;
   Make an entry (*this_label*, <LC>, –) in SYMTAB.
   (b) If an LTORG statement then
   (i) If POOLTAB [*pooltab_ptr*].*# literals* > 0 then
   Process the entries LITTAB [POOLTAB [*pooltab_ptr*].*first*] ... LITTAB [*littab_ptr*−1] to allocate memory to the literal, put address of the allocated memory area in the *address* field of the LITTAB entry, and update the address contained in location counter accordingly.
   (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (iii) POOLTAB [*pooltab_ptr*].*first* := *littab_ptr*;
   POOLTAB [*pooltab_ptr*].*# literals* := 0;

(c) If a START or ORIGIN statement then
LC := value specified in operand field;

(d) If an EQU statement then

(i) *this_addr* := value of <*address specification*>;

(ii) Correct the SYMTAB entry for *this_label* to (*this_label*, *this_addr*, 1).

(e) If a declaration statement then

(i) Invoke the routine whose id is mentioned in the *mnemonic info* field. This routine returns *code* and *size*.

(ii) If a symbol is present in the label field, correct the symtab entry for *this_label* to (*this_label*, <LC>, *size*).

(iii) LC := LC + *size*;

(iv) Generate intermediate code for the declaration statement.

(f) If an imperative statement then

(i) *code* := machine opcode from the *mnemonic info* field of OPTAB;

(ii) LC := LC + instruction length from the *mnemonic info* field of OPTAB;

(iii) If operand is a literal then
this_literal := literal in operand field;
if POOLTAB [*pooltab_ptr*]. *# literals* = 0 or *this_literal* does not match any literal in the range LITTAB [POOLTAB [*pooltab_ptr*]] .*first* ... LITTAB [*littab_ptr*−1] then
LITTAB [*littab_ptr*]. *value* := *this_literal*;
POOLTAB [*pooltab_ptr*]. *# literals* :=
POOLTAB [*pooltab_ptr*]. *# literals* +1;
littab_ptr := *littab_ptr* + 1;
else (i.e., operand is a symbol)
this_entry := SYMTAB entry number of operand;
Generate intermediate code for the imperative statement.

3. (Processing of the `END` statement)

(a) Perform actions (i)–(iii) of Step 2(b).

(b) Generate intermediate code for the `END` statement.

3.4.3 Intermediate Code Forms

In Section 2.3 processing efficiency and memory economy were mentioned as two criteria for the choice of intermediate code. In this section we consider some variants of intermediate codes for use in a two-pass assembler and compare them on the basis of these criteria.

The intermediate code consists of a sequence of *intermediate code units* (IC

units). Each IC unit consists of the following three fields (see Figure 3.10):

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

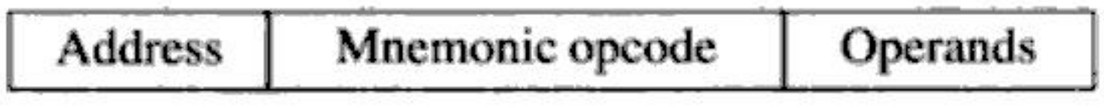


Figure 3.10 Format of an IC unit

Variant forms of intermediate codes, specifically the operand and address fields, are used in practice due to the trade-off between processing efficiency and memory economy. We discuss these variants in separate sections dealing with the representation of imperative statements, and declaration statements and directives, respectively. The information in the mnemonic opcode field is assumed to have the same representation in all the variants.

Mnemonic opcode field

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

Figure 3.11 Codes for declaration statements and directives

The mnemonic opcode 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 3.11 shows the codes for various declaration statements and assembler directives.

3.4.4 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 and mnemonic opcode fields are assumed to contain identical information in both variants.

Variant I

Figure 3.12 shows an assembly program and its intermediate code using Variant I. The first operand in an assembly statement is represented by a single digit number which is either a code in the range 1...4 that represents a CPU register, where 1 represents AREG, 2 represents BREG, etc., or the condition code itself, which is in the range 1...6 and has the meanings described in Section 3.1. 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. For a constant, the *code* field contains the representation of the constant itself. For example, in Figure 3.12 the operand descriptor for the statement START 200 is (C, 200). For a symbol or literal, the *code* field contains the entry number of the operand 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)
        ...                     ...
```

Figure 3.12 Intermediate code - Variant I

This method of representing symbolic operands requires a change in our strategy for SYMTAB management. We have so far assumed that a SYMTAB entry is made for a symbol only when its definition is encountered in the program, i.e., when the symbol occurs in the label field of an assembly statement. However, while processing a forward reference

```
MOVER AREG, A
```

it would be necessary to enter A in SYMTAB, say in entry number *n*, so that it can be represented by (S, *n*) in the intermediate code. At this point, the *address* and *length* fields of A's entry cannot be filled in. Therefore, two kinds of entries may exist in SYMTAB at any time—for defined symbols and for forward references. This fact is important for use during error detection (see Section 3.4.7).

Variant II

Figure 3.13 shows an assembly program and its intermediate code using Variant II. This variant differs from Variant I in that the operand field of the intermediate code may be either in the processed form as in Variant I, or in the source form itself. For a declarative statement or an assembler directive, the operand field has to be processed in the first pass to support LC processing. Hence the operand field of its intermediate code would contain the processed form of the operand. For imperative statements, the operand field is processed to identify literal references and enter them in the LITTAB. Hence operands that are literals are represented as (L, *m*) in the intermediate code. There is no reason why symbolic references in operand fields of imperative statements should be processed during Pass I, so they are put in the source form itself in the intermediate code.

```
        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)
        ...                     ...
```

Figure 3.13 Intermediate code - Variant II

Comparison of the variants

Variant I of the intermediate code appears to require extra work in Pass I because operand fields have to be completely processed in Pass I. However, this processing simplifies the tasks of Pass II considerably. A look at the intermediate code of Figure 3.12 confirms that functions of Pass II would be quite trivial. To process the operand field of a declaration statement, it would only need to refer to the appropriate entry in a table and use the address found there. Declaration statements such as the `DC`, `DS` and `START` statements would not require any processing at all, while statements like the `LTORG` statement would require marginal processing. The intermediate code is quite compact—it can be as compact as the target code itself if each operand reference like (S, *n*) can be represented by using the same number of bits as used for an operand address in a machine instruction.

Variant II reduces the work of Pass I by transferring the task of operand processing for some kinds of source statements to Pass II. The intermediate code is less compact because the memory operand of a typical imperative statement is represented in the source form itself. On the other hand, by transferring some tasks to

Pass II, the functions and memory requirements of the two-passes would be better balanced. Figure 3.14 illustrates its benefits. Part (a) of Figure 3.14 shows memory utilization by an assembler that uses Variant I. Data structures such as the symbol table constructed by Pass I are passed in memory while the intermediate code 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 Figure 3.14 shows memory utilization when Variant II is used. The code sizes of the two passes are now comparable, hence the overall memory requirement of the assembler is lower.

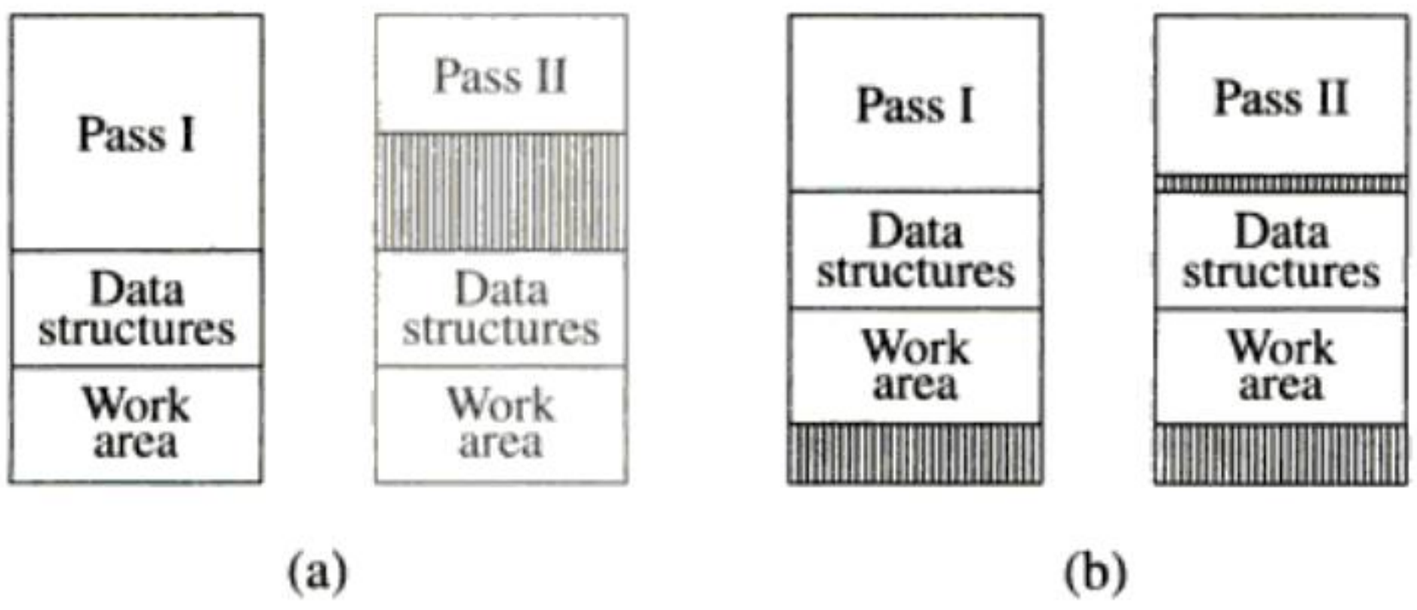


Figure 3.14 Memory requirements using (a) Variant I, (b) Variant II

Variant II is particularly well-suited if expressions can be used in 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 intermediate code. Use of this variant 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.

3.4.5 Processing of Declarations and Assembler Directives

We discuss alternative ways of processing declaration statements and assembler directives, and their comparative benefits. Two key questions in this context are:

1. Is it necessary to represent the address of each source statement in the intermediate code?
2. Is it necessary to have a representation of DS statements and assembler directives in the intermediate code?

Consider a fragment of an assembly language program 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 the intermediate code, since the effect of these statements is implied in the fact that the DC statement has the address 220. If the intermediate code does not contain the address of each source statement, a representation for the DS statements and assembler directives would have been necessary. Now, Pass II would have to determine the address for the symbol SIZE by 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.

DC statement

A DC statement must be represented in the intermediate code. 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 representation in which it should appear in the target program. 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 intermediate code.

START and ORIGIN

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

LTORG

Pass I of the assembler checks for the presence of a literal reference in the operand field of a statement. If one exists, it enters the literal in the current literal pool in LITTAB, unless a matching literal has been already entered in the current pool. When an LTORG statement appears in the source program, it allocates memory to each literal in the current pool by using the address contained in the location counter and enters the memory address in the *address* field of its LITTAB entry. It also updates the address contained in the location counter appropriately.

Once this fundamental action is performed, two alternatives exist concerning subsequent processing of literals. Pass I could simply construct an IC unit for the LTORG statement and leave all subsequent processing of literals to Pass II. Pass II would have to insert values of literals of a pool in the target program when it encounters the IC unit for an LTORG statement. This action would involve use of POOLTAB

and LITTAB in a manner analogous to Pass I, so these tables should form a part of intermediate representation of a program. Example 3.8 illustrates these actions in the processing of the program of Figure 3.8.

Example 3.8 (Processing of literals by using an IC unit for LTORG statement) Figure 3.9 showed the LITTAB and POOLTAB for the program of Figure 3.8 at the end of Pass I. Pass II would copy literals of the first pool into the target program when the IC unit for the LTORG statement is encountered. It would copy literals of the second pool into the target program when the IC unit for END is processed.

Pass I could have itself copied out the literals of the pool into the intermediate code. This action would avoid duplication of Pass I processing in Pass II and also eliminate the need to have an IC unit for an LTORG statement. The intermediate code for a literal can be made identical to the intermediate code for a DC statement so that a literal would not require any special handling in Pass II. Example 3.9 illustrates this arrangement.

Example 3.9 (Processing of literals without using an IC unit for LTORG statement) Figure 3.15 shows the IC for the first half of the program of Figure 3.8. The literals of the first pool (see Figure 3.9) are copied into the intermediate code when the first LTORG statement is encountered. Note that the opcode field of the IC units is (DL, 01), which is the opcode for the DC statement.

```
        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)
          :
```

Figure 3.15 Copying of literal values into intermediate code by Pass I

However, this alternative increases the tasks to be performed by Pass I, consequently increasing its size. It might lead to an unbalanced pass structure for the assembler with the consequences illustrated in Figure 3.14. Secondly, the literals have to exist in two forms simultaneously, in the LITTAB along with the address information, and also in the intermediate code.

3.4.6 Pass II of the Assembler

Algorithm 3.2 is the algorithm for assembler Pass II. Important data structures used by it are:

SYMTAB, LITTAB and POOLTAB

LC	:	Location counter
littab_ptr	:	Points to an entry in LITTAB
pooltab_ptr	:	Points to an entry in POOLTAB
machine_code_buffer	:	Area for constructing code for one statement
code_area	:	Area for assembling the target program
code_area_address	:	Contains address of *code_area*

Algorithm 3.2 (Second pass of a two-pass assembler)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   LC := 0;
2. While the next statement is not an END statement
   (a) Clear *machine_code_buffer*;
   (b) If an LTORG statement
      (i) If POOLTAB [*pooltab_ptr*]. *# literals* > 0 then
         Process literals in the entries LITTAB [POOLTAB [*pooltab_ptr*] .*first*] ... LITTAB [POOLTAB [*pooltab_ptr*+1]−1] similar to processing of constants in a DC statement. It results in assembling 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
      (i) LC := 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 the memory area required by the declaration statement;
   (e) If an imperative statement
      (i) Get address of the operand from its entry in SYMTAB or LITTAB, as the case may be.
      (ii) Assemble the instruction in *machine_code_buffer*.
      (iii) *size* := size of the instruction;
   (f) If *size* ≠ 0 then
      (i) Move contents of *machine_code_buffer* to the memory word with the address *code_area_address* + <LC>;

(ii) LC := LC + *size*;

3. (Processing of the END statement)
   (a) Perform actions (i)–(iii) of Step 2(b).
   (b) Perform actions (i)–(ii) of Step 2(f).
   (c) Write *code_area* into the output file.

Output interface of the assembler

Algorithm 3.2 produces a target program which is in the machine language of the target computer. However, most assemblers produce the target program in the form of an *object module*, which is processed by a linkage editor or loader to produce a machine language program (see Figure 2.10). The information contained in object modules is discussed in Chapter 5.

3.4.7 Program Listing and Error Reporting

It is conventional for an assembler to produce a program listing that shows a source statement and the target code, if any, generated for it. The listing also reports any errors that the assembler might have found in the program. Error reporting is most effective when the listing shows an error against an erroneous statement itself.

Design of an error indication scheme involves some decisions that 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 until Pass II. It conserves memory and avoids some amount of duplicate processing. However, a listing produced in Pass I can report only certain errors against the source statement. 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 symbols that are not defined in the program become known only after the complete source program has been processed, hence they cannot be indicated against the statements that contained them. The target code can be printed later in Pass II; however, it is important to show which parts of the target code correspond to which source statements. Example 3.10 illustrates an arrangement to show this correspondence in the listing.

Example 3.10 (Error reporting in Pass I of the assembler) Figure 3.16 shows a program listing that is produced 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 the OPTAB. In Statement 21, `A` is known to be a duplicate definition because an entry for `A` already exists in the symbol table. However, use of the undefined symbol `B` in Statement 10 is harder to detect because at the end of Pass I we would have no record that a forward reference to `B` existed in Statement 10. This problem can be resolved by making an entry for `B` in the symbol table while processing

```
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 ** Use of undefined symbol B in statement 10
```

Figure 3.16 Error reporting in Pass I

Statement 10. This entry would indicate that a forward reference to B exists in Statement 10. All such entries would be processed at the end of Pass I to check whether a definition of the symbol has been encountered. If not, the symbol table entry would contain sufficient information for error reporting. Note that the target code could not be included in the listing because it has not yet been generated. To provide a cross reference between a source statement and the target code that corresponds to it, the memory address of the code is printed against the statement.

For effective error reporting, it is necessary to report each error against the erroneous statement itself. It can be achieved by delaying program listing and error reporting actions until Pass II. Now the errors in a statement and the target code, if any, that corresponds to it can be printed against the source statement itself. Example 3.11 illustrates such a listing.

Example 3.11 (Error reporting in Pass II of the assembler) Pass II of the assembler performs error reporting actions in addition to the actions related to location counter processing and building of the symbol table shown in Algorithm 3.2. Figure 3.17 contains a program listing produced in Pass II. Indication of errors in Statements 9 and 21 was as easy as in Example 3.10. Indication of the error in Statement 10 was equally easy—the symbol table was searched for an entry of the symbol B that appeared in the operand field and an error was reported because no matching entry was found. Note that the target code for each source statement appears against it in the listing.

3.4.8 Some Organizational Issues

The schematic of Figure 3.18 shows the data structures and files used in a two-pass assembler. We discuss how each of them should be stored and accessed in

```
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 ** Use of 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
```

Figure 3.17 Error reporting in Pass II

the assembler passes.

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

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 intermediate code generated for it would also be written into another file. The target code and the program listing can be written out as separate files by Pass II. Since all these files are sequential in nature, it is beneficial to use the techniques of blocking and buffering of records discussed later in Chapter 13.

3.5 A SINGLE-PASS ASSEMBLER FOR INTEL x86 FAMILY PROCESSORS

Processors of the Intel x86 family are downward compatible with the Intel 8088 processor that was used in the IBM PC. A key difference between the architecture of the Intel x86 processors and the CPU of the hypothetical computer of previous sections is the use of segment-based addressing of memory. We focus on the addressing issues that arise in segment-based memory addressing and describe how a single-pass assembler handles the forward reference problem in this environment.

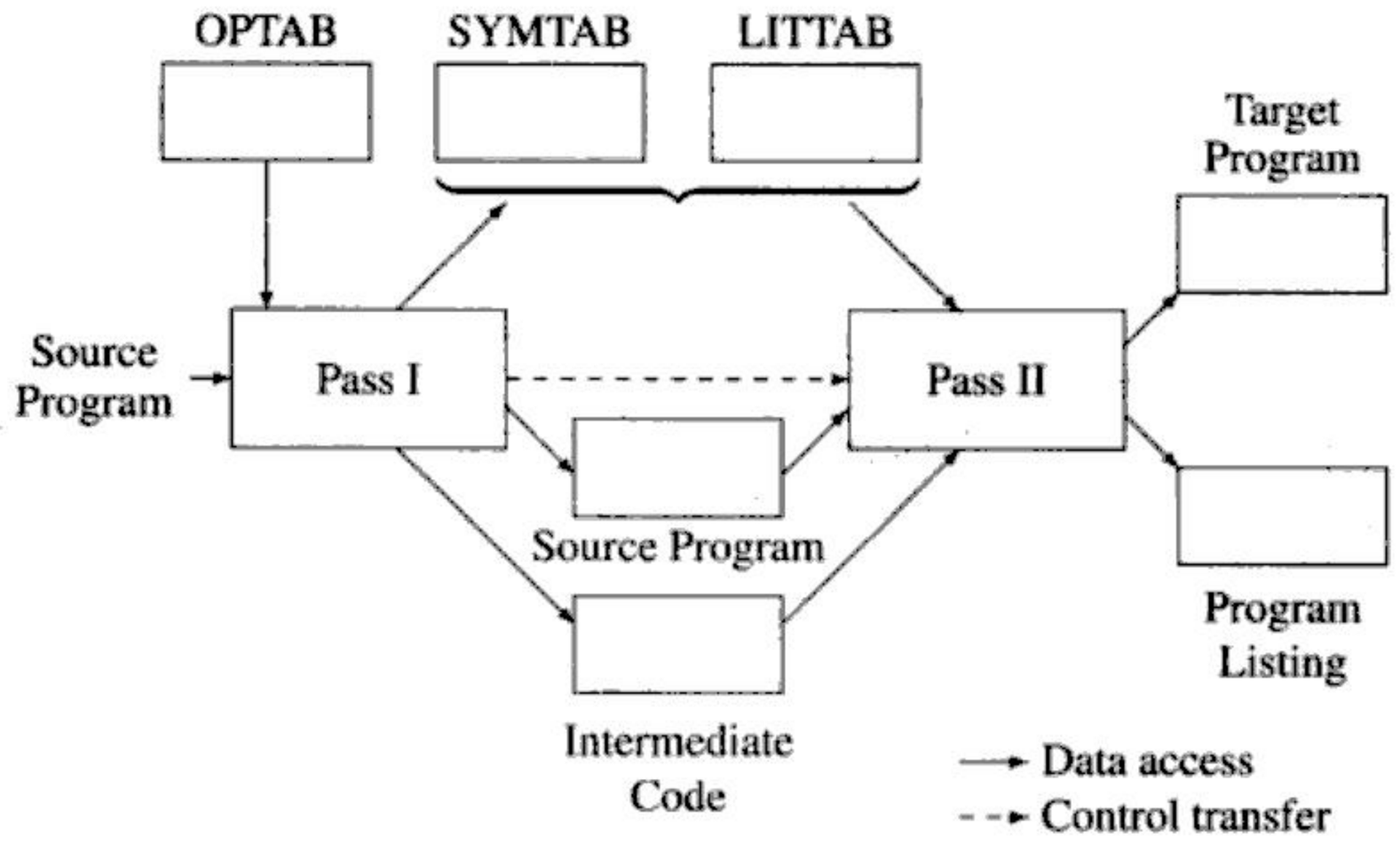


Figure 3.18 Use of data structures and files in a two-pass assembler

3.5.1 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 Figure 3.19):

- 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 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 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. The architecture supports stacks for storing subroutine and interrupt return addresses, parameters, and other data. Two stack pointer registers called SP and BP are provided for addressing stacks; push and pop instructions operate on them. Register SP points into the stack implicitly used by the architecture to store subroutine and interrupt return addresses. Register BP can be used by the programmer in any desired manner.

Memory is not accessed through absolute addresses as assumed in previous sections but through addresses contained in *segment registers*. A program may consist of many components called *segments*, where each segment may contain a part of the program's code, data, or stack. Addresses of any four of these segments may be contained in the four segment registers provided in the CPU. The Code segment

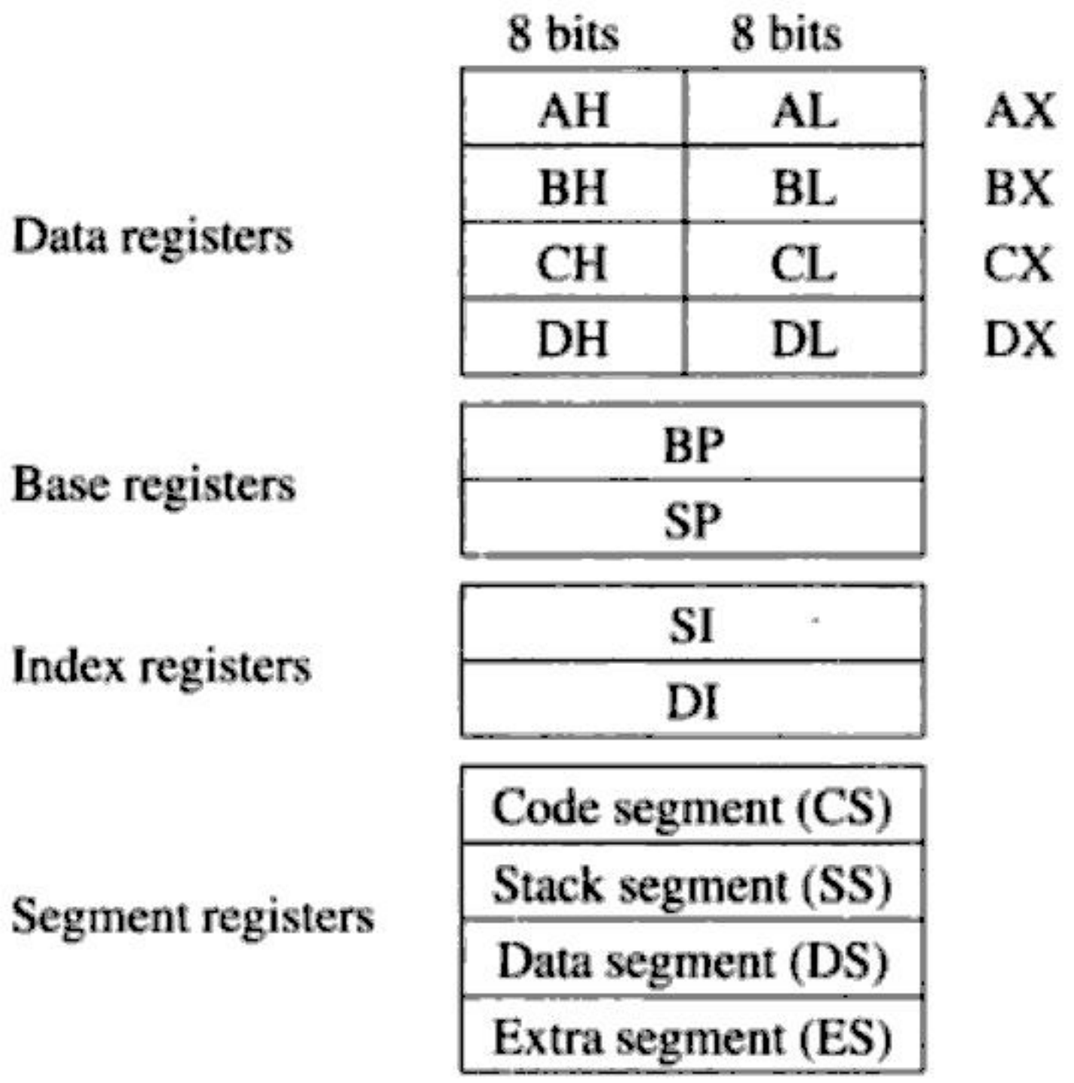


Figure 3.19 Registers of the Intel 8088 processor

(CS), Data segment (DS), and Stack segment (SS) registers are typically used to contain the start addresses of a program's code, data, and stack, respectively. The Extra segment (ES) register can be used to contain the address of any other memory area. Each segment register is 16 bits in size. An instruction uses a segment register and a 16 bit offset to address a memory operand. The absolute address of the operand is computed as follows: The address contained in the segment register is extended by adding four lower order zeroes to obtain a 20-bit *segment base address*, also simply called the *segment base*, which is the address of the memory location occupied by the first byte of the segment. The offset is now added to it to obtain a 20 bit memory address. This way, the size of each segment is limited to 2^{16} bytes, i.e., 64 Kbytes; however, memory can have a size of 2^{20} bytes, i.e., 1 MB. Segment-based addressing makes it possible to *relocate* a program, that is, change the memory area used for its execution, easily. When such a change is to be made, it is enough to simply change the addresses loaded in the segment registers. The memory addresses used during execution of the program would now automatically lie in the new memory area occupied by the program.

The 8088 architecture provides 24 addressing modes. These are summarized in Figure 3.20. 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 displacement which is taken to be an offset from the segment base contained in a segment register. The segment register may be explicitly indicated in a prefix of the instruction; otherwise,

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-and-indexed	`MOV SUM, 56H [SI] [BX]`	Data disp. = 56H + (SI) + (BX)

Figure 3.20 Addressing modes of 8088 ('(..)' implies 'contents of')

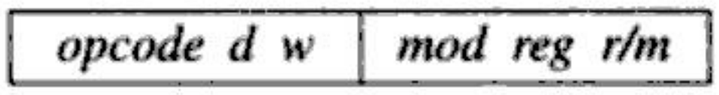
a default segment register is used. In the *indexed mode*, contents of the index register (SI or DI) indicated in the instruction are added to the 8 or 16 bit displacement contained in the instruction. The result is taken to be the offset from the segment base of the data segment. In the *based mode*,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

3.5.2 Intel 8088 Instructions

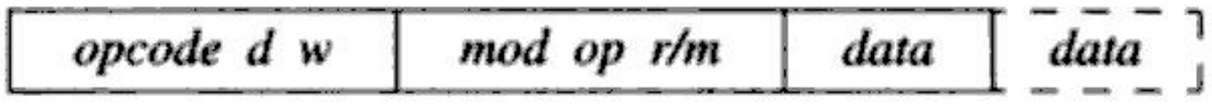
Arithmetic instructions

The operands of arithmetic instructions can be in one of the four 16 bit registers, or in a memory location designated by one of the 24 addressing modes. The arithmetic instructions can be in one of the three instruction formats shown in Figure 3.21. The *mod* and *r/m* fields specify the first operand, whose value can be in a register or in memory, while the *reg* field describes the second operand, whose value is always in 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; otherwise, 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 table in the middle part of Figure 3.21 indicates the conventions used for determining the first operand. If *mod* = 00, 01 or 10 the first operand is in memory; otherwise, it is in a register. The table in the lower part of the figure indicates the conventions for determining the second operand, which is always in memory.

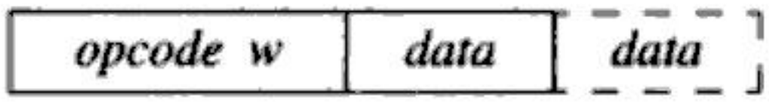
(a) *Register/Memory to Register*



(b) *Immediate to Register/Memory*



(c) *Immediate to Accumulator*



r/m	*mod* = 00	*mod* = 01 Note 1	*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 3	(BP) + d8	Note 2	DH	SI
111	(BX)	(BX) + d8	Note 2	BH	DI

Note 1 : d8 denotes an 8-bit displacement
Note 2 : Same as in the previous column, except d16 instead of d8
Note 3 : (BP) + DISP for indirect addressing, d16 for direct

reg	*Register*	
	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

Figure 3.21 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

Figure 3.22 Sample assembly statements and corresponding instructions of 8088

Figure 3.22 contains some assembly statements and the corresponding instructions. Note that in the first assembly statement, AL could be encoded into the first or the second operand of the instruction, and the *d* field could be set accordingly. In the second statement, however, AL has to be encoded into second operand because `12H [SI]`, which denotes a displacement of 12_{16} from address contained in the SI register, has to be in the first operand. Here, *mod* = 01 since only one byte of displacement is adequate. The third statement 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. In situations such as these, the assembler has to analyze the available options and determine the best instruction.

Segment overrides

For operands in arithmetic and `MOV` instructions, the architecture uses the data segment by default. To use any other segment, an instruction has to be preceded by a 1-byte *segment override prefix*, which has the format 001 *seg* 110 where *seg*, represented in 2 bits, has the meanings shown in Figure 3.23.

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

Figure 3.23 Segment codes for use in the segment override prefix

Example 3.12 (Using the segment override prefix) In Figure 3.22, if the code segment is to be used instead of the data segment in the second statement, the statement would have to be rewritten as follows:

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

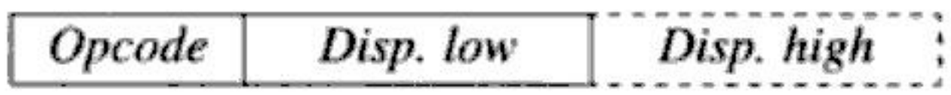
The assembler would encode this statement as

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

Control transfer instructions

Figure 3.24 shows formats of control transfer instructions. Control may be transferred to an address within the same segment, or to an address in another segment. Where possible, an intra-segment transfer is assembled using a self-relative displacement in the range of −128 to +127 because it leads to a shorter instruction. In other situations, an intra-segment transfer is assembled using a 16 bit offset within the segment. An inter-segment transfer requires a new segment base address and an offset to be provided. Its execution involves loading of the segment base into the CS segment register, which makes it slower than an intra-segment transfer. Control transfers can be both direct and indirect.

(a) *Intrasegment*



(b) *Intersegment*

(c) *Indirect*

Figure 3.24 Formats of control transfer instructions

A subroutine call instruction pushes the offset of the next instruction within the code segment on the stack. This offset is used to return control to the calling program. In the case of an inter-segment subroutine call, the address in the CS segment register is first pushed on the stack, followed by the offset of the next instruction.

3.5.3 The Assembly Language of Intel 8088

Statement format

The format of the assembly language 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 3.22 contained 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 such parentheses. A segment override is specified in the operand to which it applies, e.g., `CS:12H [SI]` indicates that the operand exists in the code segment.

Assembler directives

The `ORG`, `EQU` and `END` directives are analogous to the `ORIGIN`, `EQU` and `END` directives described in Sections 3.4 and 3.1. The start directive is not supported because the `ORG` directive subsumes its functionality. The concept of literals, and the `LTORG` directive, are redundant because the 8088 architecture supports immediate operands.

Declarations

The Intel assembly language provides directives that both reserve memory and initialize it to desired values. Thus, these directives combine the functions of the `DC` and `DS` directives of section 3.1.1. The `DB`, `DW` and `DD` directives have the following meanings:

```
A       DB      25          ;Reserve a byte & initialize it
B       DW      ?           ;Reserve a word, but do not initialize
ADD_A   DW      A           ;Reserve & initialize word to A's offset
C       DD      6DUP(0)     ;6 Double words, all initialized to 0s
```

The directives `DQ` and `DT` reserve and initialize a quad-word, which is an area of 8 bytes whose start address is aligned on a multiple of 8, and ten bytes, respectively.

`EQU` *and* `PURGE`

As described in Section 3.4, the `EQU` directive defines a symbolic name to represent either a value or another symbolic name. The name so defined can be 'undefined' through a `PURGE` directive. A purged name can be reused for other purposes later in the program. Example 3.13 illustrates use of this directive.

Example 3.13 (The PURGE directive) Following program illustrates use of the `EQU` and `PURGE` directives.

```
XYZ         DB          ?
ABC         EQU         XYZ   ; ABC represents the name XYZ
            PURGE       ABC   ; ABC no longer represents XYZ
ABC         EQU         25    ; ABC now represents the value 25
```

`SEGMENT`, `ENDS` *and* `ASSUME`

An assembly language program is a collection of *segments*. The address of a memory operand in an Intel 8088 instruction is specified as a pair (*segment register*, *offset*),

where the segment register contains the address of one of the segments of the program. Thus, for assembling the reference to the symbol ALPHA in the following statement:

```
ADD          AL, ALPHA
```

the assembler has to determine the offset of ALPHA from the address contained in a segment register. Hence the assembler must know which segment contains the statement that defined ALPHA, and which segment register would contain the address of that segment when the ADD statement is executed. The SEGMENT and ENDS directives indicate the beginning and end of a segment, respectively. They enable the assembler to determine which segment contains the definition of ALPHA. The ASSUME directive has the following syntax:

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

It informs the assembler that the address of the indicated segment would be present in *<segment register>* when the part of the program that begins with this statement is executed. Note that it is the program's responsibility to actually load the segment's address into the specific segment register during its execution. The directive ASSUME *<register>* : NOTHING nullifies the effect of the previous ASSUME statement for *<register>*.

Note that a memory operand in an instruction is *addressable* only if the address of the segment that contains the operand is present in one of the segment registers. The assembler must indicate an error if a memory operand is not addressable. Example 3.14 illustrates how the assembler uses the information in the SEGMENT, ENDS and ASSUME directives for determining the segment register and offset for accessing a memory operand.

Example 3.14 (Assembling of memory operands) 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. The segment SAMPLE_DATA contains the storage reserved through the two DW directives, while the segment SAMPLE_CODE contains

the code corresponding to the rest of the statements. The ASSUME directive informs the assembler that the start address of SAMPLE_DATA would be contained in the DS register. While assembling the statement MOV AX, SUM the assembler first computes the offset of SUM from the start of the segment that contains it, which in this case is the SAMPLE_DATA segment. This offset is 200 bytes because the SAMPLE_DATA segment contains 100 words before the area named SUM. The assembler now checks whether the segment in which SUM exists is addressable at the current place in the program, that is, whether the address of the segment would be contained in one of the segment registers when the MOV statement is executed. Following from the ASSUME statement, it knows that the DS register would contain the address of segment SAMPLE_DATA, so it encodes SUM to be an offset of 200 bytes from the DS register. Note that the program must load the correct address into the DS register before executing this reference to SUM. If the ASSUME directive were ASSUME ES:SAMPLE_DATA, the assembler would have generated a segment override prefix while assembling the operand SUM in the statement MOV AX, SUM. Now the program must load the correct address into the ES register.

PROC, ENDP, NEAR *and* FAR

The PROC and ENDP directives 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 a *far* call. Parameters for the called procedure can be passed either through registers or on the stack. Example 3.15 illustrates assembling of a call on a far procedure.

Example 3.15 (Far procedures) 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
```

CALCULATE is declared as a far procedure, so it need not be addressable at the CALL statement. The assembler encodes the CALL statement by using a far instruction that has the segment base of the segment that contains the procedure CALCULATE and the offset of CALCULATE within that segment in the operand field.

PUBLIC *and* EXTRN

A symbol that is declared in one assembly program can be accessed only from within that program. If the symbol is to be accessed in other programs, it should be specified

in a PUBLIC directive. Any other program wishing to use this symbol must specify it in an EXTRN directive, which has the syntax

```
EXTRN        <symbolic name> : <type>
```

As we shall see in Chapter 5, the linker uses the information specified in the PUBLIC and EXTRN directives for linking a program with other programs. The type information provided in the EXTRN directive is used by the assembler in conjunction with the analytic operators. For labels of DC and DS statements, the type can be a word, a byte, etc. For labels of instructions, the type can be FAR or NEAR.

Analytic operators

An analytic operator either provides components of a memory address, or provides information regarding the type and memory requirements of operands. The SEG and OFFSET operators provide the segment and offset components of the memory address of an operand. The TYPE operator provides a numeric code that indicates the manner in which an operand is defined; the codes are 1 (byte), 2 (word), 4 (double word), 8 (quad-word), 10 (ten bytes), −1 (near instruction) and −2 (far instruction). The SIZE operator indicates the number of units in an operand, while The LENGTH operator indicates the number of bytes allocated to the operand.

Example 3.16 (Analytic operators) Consider the fragment of an assembly program:

```
                MOV         AX, OFFSET ABC
BUFFER          DW          100 DUP (0)
```

The MOV statement loads the offset of the symbol ABC within its segment into the AX register. BUFFER has the TYPE of 2, SIZE of 100, and LENGTH of 200 bytes. These operators can be used in MOV statements such as

```
MOV         CX, LENGTH XYZ
```

which loads the length of XYZ into the CX register.

Synthetic operators

A programmer may wish to access the same memory area as operands of different types. For example, she may wish to access the first word in an operand of type quad-word as an operand of type 'word', or access its first byte as an operand of type 'byte'. To facilitate such usage, the PTR operator provides a method to define a new memory operand that has the same segment and offset components of address as an existing memory operand, but has a different type. No memory allocation is implied by its use. The THIS operator is analogous—it defines the new memory operand to have the same address as the *next* byte in the program. Hence its effect is sensitive to its placement in the program.

Example 3.17 (The PTR and THIS operators) 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 that has the same address as XYZ, while FAR_LOOP is a FAR symbolic name that has the same address as LOOP. Thus, while JMP LOOP would be assembled as a near jump, JMP FAR_LOOP would be assembled as a far jump. Exactly the same effect could have been achieved by using the THIS operator 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
```

Here FAR_LOOP has the same address as LOOP because it immediately precedes the statement that defines LOOP.

3.5.4 Problems of Single-Pass Assembly

The forward reference problem faced in single-pass assembly is aggravated by the nature of the 8088 architecture. We discuss two aspects of this issue using the program of Figure 3.25.

Forward references

A symbolic name may be forward referenced in many different ways. Assembly is straightforward when the forward referenced symbol is used as a data operand in a statement. As discussed in Section 3.3, an entry can be made in the table of incomplete instructions (TII) when the forward reference is encountered. It 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 analyzed to complete the instruction. However, the 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 be assembled into instructions of different formats and 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, whether the jump is near or far would not be known until sometime

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		

Figure 3.25 An assembly program of Intel 8088 for illustrating problems in single-pass assembly

later in the assembly process! This problem is solved by assembling such instructions with a 16 bit offset unless the programmer uses the keyword `SHORT` to indicate that a short displacement would suffice, e.g., as in `JMP SHORT LOOP`. The program of Figure 3.25 contains a forward reference in the statement `JNE NEXT`. However, the above problem does not arise here because the mnemonic `JNE` indicates 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. It may influence the size and length of a memory area, in turn affecting memory allocation and addresses assigned to symbols. Such usage will have to be disallowed to facilitate single-pass assembly. Example 3.18 illustrates this aspect.

Example 3.18 (Difficulties in single-pass assembly concerning synthetic operators) Consider the statements

```
XYZ     DB      LENGTH ABC DUP(0)
        ...
ABC     DD      ?
```

Here, the size of XYZ cannot be determined in a single-pass due to the forward reference to ABC.

Use of segment registers

When the assembler encounters a forward reference to a symbol *symb* in a statement, it needs to determine what offset and what segment register it should use in the operand field of the corresponding instruction. It can determine the information as follows: When the assembler encounters an ASSUME statement

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

it can store the pair (*segment register*, *segment name*) in a *segment registers table* (SRTAB). For handling the reference to a symbol *symb* in an assembly statement, the assembler would access the symbol table entry of *symb* and find seg_{symb}, which is the name of the segment that contains the definition of *symb*. Using this information it would form the pair (seg_{symb}, $offset_{symb}$). It would then access the information stored in SRTAB to find which register would contain the address of seg_{symb}. Let it be register *r*. It would synthesize the pair (*r*, $offset_{symb}$) and put it in the address field of the target instruction.

However, this strategy would not work while assembling forward references. Consider Statements 6 and 13 in Figure 3.25 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 the SRTAB indicated that the DS register would contain the segment base of DATA. However, the SRTAB does not contain that information presently; it contains the pair (ES, DATA) because of the statement ASSUME ES:DATA which appeared as the 8^{th} statement in the program. A similar problem may arise while assembling forward references contained in branch instructions. To handle this problem the old information in SRTAB should be preserved while processing an ASSUME statement. Accordingly, the following provisions are made:

1. A new SRTAB is created when an ASSUME statement is encountered. This SRTAB differs from the old SRTAB only in the entries for the segment register(s) named in the ASSUME statement. Since many SRTABs may exist at any time, an array named SRTAB_ARRAY is used to store the SRTABs. An individual SRTAB is referred to by its entry number in SRTAB_ARRAY.
2. Instead of using the TII, a *forward reference table* (FRT) is used. Each entry of the FRT contains the following information:
   (a) The address of the instruction whose operand field contains the forward reference
   (b) The symbol to which the forward reference is made

(c) The kind of reference (e.g., T : analytic operator TYPE, D : data address, S : self relative address, L : length, F : offset, etc.)

(d) Identity of the SRTAB that should be used for assembling the reference.

Example 3.19 illustrates how these provisions are adequate for handling the problem concerning forward references mentioned earlier.

Example 3.19 (Difficulties in single-pass assembly concerning segment registers) Two SRTABs would be built for the program of Figure 3.25. SRTAB #1 is built while processing the 2^{nd} statement. It contains the pairs (CS, CODE) and (DS, DATA). Statement 6 contains a forward reference to COUNT, hence the entry (008, COUNT, D, SRTAB #1) is made in the FRT. SRTAB #2, which is built while processing the 8^{th} statement, contains the pairs (CS, CODE) and (ES, DATA). 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 segment register.)

3.5.5 Design of the Assembler

The algorithm for the Intel 8088 assembler is given at the end of this section as Algorithm 3.3. LC processing in this algorithm differs from LC processing in the first pass of a two-pass assembler (see Algorithm 3.1) in one significant respect. In Intel 8088, the unit for memory allocation is a byte; however, certain entities require their first 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 declarations and imperative statements, the assembler first aligns the address contained in the LC on the appropriate boundary. We call this action *LC alignment*. Allocation of memory for a statement is performed after LC alignment.

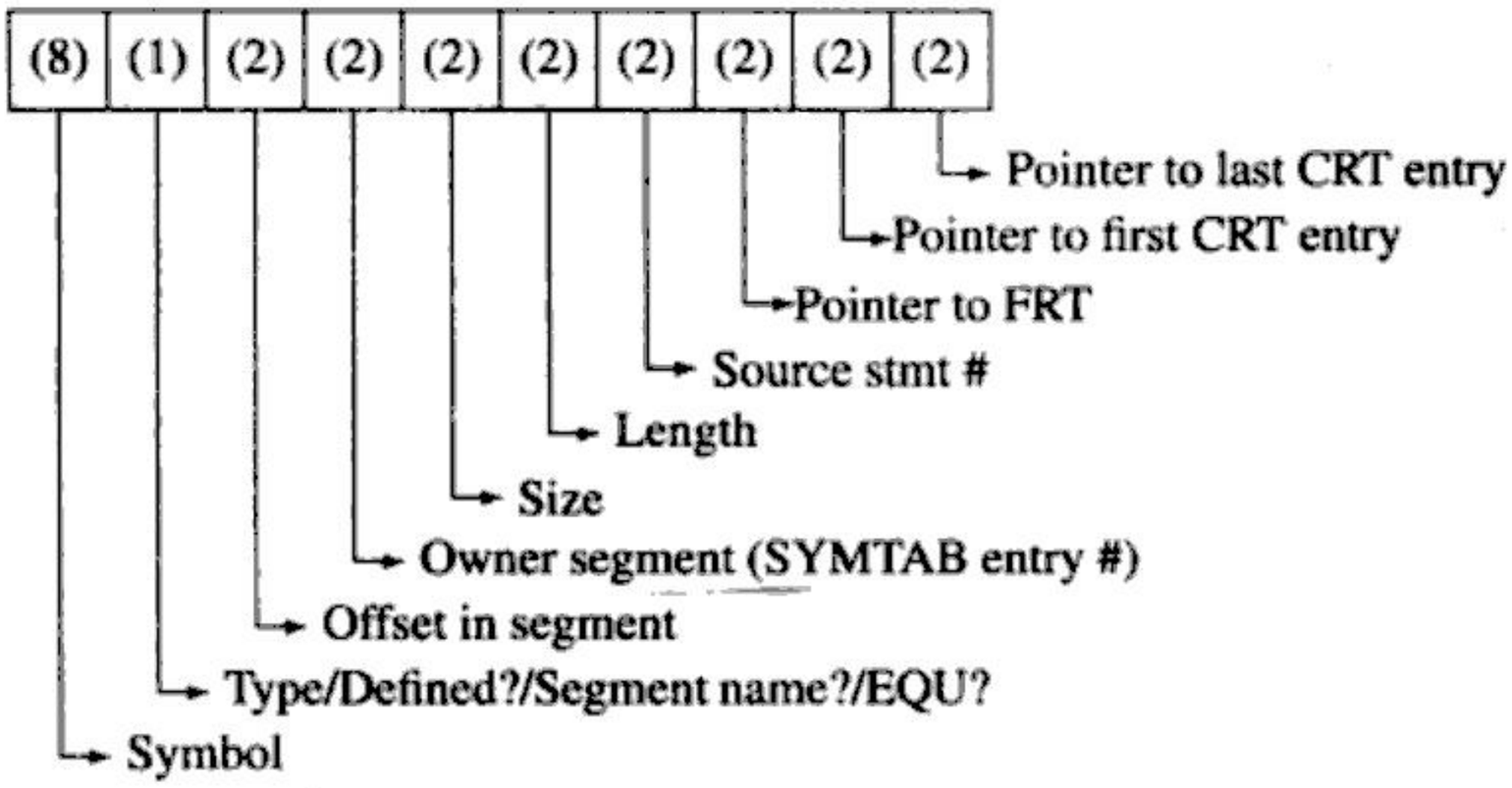
The data structures of the assembler are illustrated in Figure 3.26, where a number in parentheses indicates the number of bytes required for a field. Details of the data structures are as follows:

- 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 handles that opcode. *Alignment/format info* is specific to a given routine. For example, in Figure 3.26 the code '00H' in the *Alignment/format info* field in the entry for the opcode JNE implies that routine R2 should use the instruction format with self-relative displacement. If the code 'FFH' were to be used, it would imply that all instruction formats are to be supported, so the routine must decide which machine opcode to use.

(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)	*SYMTAB entry #* (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)

Figure 3.26 Data structures of the assembler

- The *symbol table* (SYMTAB) is also hash-organized and contains information about symbols defined and used in the source program. The contents of some important fields are as follows: The *owner segment* field indicates the id of the segment in which the 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, the *type* field indicates whether the symbol is to be given a numeric value or a textual value, and the value itself is accommodated in the *owner segment* and *offset* fields of the entry.

- The *segment register table* (SRTAB) contains four entries, one for each segment register. Each entry shows the SYMTAB entry # of the segment whose address is contained in the segment register. SRTAB_ARRAY is an array of SRTABs.

- The *forward reference table* (FRT) and *cross reference table* (CRT) are organized as linked lists.

Being linked lists, the sizes of FRT and CRT change as a program is processed. To avoid making a fixed commitment of storage to these tables, both of them are organized in a single memory area that grows from the high end of memory to its low end. The assembler places the generated target code from the low end of memory to its high end. This way, no size restrictions need to be placed on individual tables. Assembly of a source program would fail only if the target code overlaps with its tables.

Forward references

Information concerning forward references to a symbol is organized as a linked list. Thus, the *forward reference table* (FRT) contains a set of linked lists, one for each forward referenced symbol. The *FRT pointer* field of a SYMTAB entry points to the head of a symbol's list. Since the ordering of FRT entries in a list is not important, for efficiency reasons new entries are added at the beginning of the list. Each FRT entry contains SRTAB # of the SRTAB that is to be used for assembling the forward reference. It also contains an instruction address and a *usage code* that indicates where and how the forward reference is to be assembled. When the definition of a symbol is encountered, the FRT entries in its list of forward references (if any) are processed, and the forward references' list is discarded. To minimize the size of FRT, entries in the discarded list are reused for storing information about other forward references.

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 by statement numbers. The assembler uses the *cross reference table* (CRT) to collect the relevant information.

Each entry in the CRT describes one reference to one symbol. Entries pertaining to the same symbol are linked together. New entries are added at the end of the list so that entries would be in ascending order by statement number. The SYMTAB entry of each symbol points to the head and tail of its linked list in the CRT. Example 3.20 illustrates operation of the assembler.

Example 3.20 (Single-pass assembly of Intel 8088 programs) Figure 3.27 illustrates some of the contents of important data structures after processing Statement 19 of the source program shown in Figure 3.25. 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 should be put in the instruction. When the definition of `NEXT` was processed in Statement 14, the corresponding instruction was completed using this FRT entry and the FRT entry was discarded.

FRT entries currently exist for symbols `COUNT` and `STRNG`. Both references to `COUNT` in the source program are forward references in 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`.

Subsequent processing of the program proceeds as follows: At the end of processing Statement 20 (but before incrementing the location counter), its label, viz., `COUNT`, would be looked up in SYMTAB. An entry exists for it with *defined* = '*no*'. It implies that `COUNT` has been forward referenced. Its *segment* and *offset* fields would be set now. The forward reference chain would then be traversed and each forward reference would be processed to detect any errors in the forward reference, and to complete the machine instruction containing the forward reference. The second forward reference to `COUNT` would pass the error detection step and lead to completion of the machine instruction with offset 0024. However, the first forward reference expects a word alignment for `COUNT` (since immediate data is 2 bytes in length) which is not the case. An error would be indicated at this point. The *FRT pointer* field of `COUNT`'s SYMTAB entry would be reset and the FRT entries for `COUNT` would be destroyed.

Listing and error indication

The program listing and error reporting function faces the problems discussed in Section 3.4.7. Hence the program listing contains the statement in source form, its serial number in the program and the memory address assigned to it. The target code is printed at the end of assembly. An error pertaining to a forward reference cannot be reported against the statement that contains the forward reference because that statement would have been already listed out. If the error is simply reported at the end of the source program, it would be rather cumbersome for the programmer to find the erroneous statement. To overcome this problem, the serial number of the source statement containing a forward reference is stored in the FRT entry along with other relevant information (see Figure 3.27). If an error is detected while processing

	Symbol	D ?	S ?	T	Offset	Owner segment	Length & size	FRT pointer	CRT pointer First	CRT pointer Last
1	CODE	Y	Y							
2	DATA	N	Y							
3	COMP	Y	N	-1	19	1		-		
4	NEXT	Y	N	-1	27	1		-		
5	COUNT	N	N							
6	STRNG	N	N							

SYMTAB

Ptr	SRTAB	Addr	code	stmt #
	#1	0008	D	0006
	#2	0024	D	0013

Ptr	stmt #
	16
	12

Cross Reference Table (CRT)

	#1	0005	L	0005
	#1	0011	F	0007

Forward Reference Table (FRT)

00(ES)	
01(CS)	1
10(SS)	
11(DS)	2

SRTAB #1

00(ES)	2
01(CS)	1
10(SS)	
11(DS)	

SRTAB #2

SRTAB_ARRAY

Figure 3.27 Data structures after processing Statement 19 of the program in Figure 3.25

a forward reference to a symbol, the statement number found in the FRT entry would be included in the error report.

Example 3.21, which follows the algorithm of the single-pass assembler illustrates application of this strategy.

3.5.6 Algorithm of the Single-Pass Assembler

Algorithm 3.3 is the algorithm of the assembler. Important data structures used by it are as follows:

SYMTAB, SRTAB_ARRAY, CRT, FRT and ERRTAB

LC	:	Location counter
code_area	:	Area for assembling the target program
code_area_address	:	Contains address of *code_area*
srtab_no	:	Number of the current SRTAB
stmt_no	:	Number of the current statement
SYMTAB_segment_entry	:	SYMTAB entry # of current segment
machine_code_buffer	:	Area for constructing code for one statement

Algorithm 3.3 (Single-pass assembler for Intel 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 the next statement is not an END statement
   (a) Clear *machine_code_buffer*.
   (b) If a symbol is present in the label field then
   this_label := symbol in the label field;
   (c) If an EQU statement
   (i) *this_address* := value of <*address specification*>;
   (ii) Make an entry for *this_label* in SYMTAB with
   offset := *this_addr*;
   Defined := '*yes*';
   owner_segment := *owner_segment* in SYMTAB entry of the symbol in the operand field.
   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) For each specification in the ASSUME statement

A. *this_register* := register mentioned in the specification.

B. *this_segment* := entry number of SYMTAB entry of the segment appearing in the specification.

C. Make the entry (*this_register, this_segment*) in SRTAB_ARRAY [*srtab_no*]. (It overwrites an existing entry for *this_register*.)

D. *size* := 0;

(e) If a SEGMENT statement

(i) Make an entry for *this_label* in SYMTAB and note the entry number.

(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 the operand is a symbol *symb* then
enter *stmt_no* in CRT list of *symb*.

(ii) If the 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 *symb* in SYMTAB with *defined* := '*no*';
Make the entry (*srtab_no*, LC, *usage code, stmt_no*) in FRT of *symb*.

(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 with

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* + <LC>;

(iii) LC := LC + *size*;

(iv) Process forward references to the symbol. Check for alignment and addressability errors. Enter errors in the ERRTAB.

(v) List the statement along with errors pertaining to it found in the ERRTAB.

(vi) Clear ERRTAB.

3. (Processing of END statement)

(a) Report undefined symbols from the SYMTAB.

(b) Produce cross reference listing.

(c) Write *code_area* into the output file.

Example 3.21 (Error reporting in single-pass assembler for Intel 8088) Error in the forward reference to COUNT in Statement 6 of Figure 3.25 would be reported as follows:

Stmt no.		*Source statement*		*Offset*	*Instrn*
006		MOV	COUNT, 0000	0005	...
...		...		...	...
020	COUNT	DB	?	0001	...

** error ** Illegal forward reference (alignment) from Stmt 6.

3.6 SUMMARY

The assembly language is a machine-level programming language. Each statement in an assembly program either corresponds to a machine instruction, declares a constant, or is a *directive* to the assembler. The assembly language provides many facilities that free the programmer from having to deal with strings of 0s and 1s as instructions and data of a program. Three key facilities are use of *mnemonic operation codes* and *symbolic operands* in instructions, and facilities for declaring data and reserving memory.

An *assembler* is a language processor that converts an assembly language program into a target program that is either a machine language program or an *object module*, which is a program form used by linkage editors. In the *analysis phase*, it

analyzes the source program to note the symbolic names used in the program and the instructions or data to which they correspond. It has to perform *memory allocation* for determining the memory address of each instruction or data, so it maintains a data structure called the *location counter* (LC) to keep track of the address that the next instruction or data in the target program would have. As it processes assembly statements, it updates the location counter by the size of each instruction or data. Some assembler directives require a new address to be put in the location counter. These actions are collectively called *LC processing*. In the synthesis phase, the assembler synthesizes the target program by using the information obtained during the analysis phase. The assembler uses two key tables—it uses the *mnemonics table* to find which mnemonic operation codes correspond to which instructions, and uses the *symbol table* to store information about symbolic names used in the program.

As defined in Chapter 2, a *pass* of a language processor performs language processing functions on every statement in a source program, or in its equivalent representation. A single-pass assembler analyzes a source statement and immediately synthesizes equivalent statements in the target program. In a *multi-pass organization* of assemblers, the first pass of the assembler processes the source program to construct an *intermediate representation* that consists of the symbol table and an *intermediate code*. The second pass uses the information in the intermediate representation to synthesize the target code. Two variants of the intermediate code are discussed.

A *forward reference* in a program is the use of a symbol that precedes the statement that defines the symbol. A single-pass assembler uses the technique of *backpatching* to handle forward references. When it encounters a forward reference in a source statement, it generates a machine instruction that has a blank operand address field and enters the address of the instruction in a table of incomplete instructions. When it encounters the statement that defines the forward referenced symbol, it completes all instructions that contained forward references to the symbol. A multi-pass assembler can handle forward references more simply. The first pass puts information about all defined symbols in the the symbol table, and the second pass uses this information while generating instructions in the target program.

A case study of a single-pass assembler for processors of the Intel x86 is included. The Intel 8088 uses segment-based addressing, which raises interesting issues in the specification of operand addresses in machine instructions and in the handling of forward references.

TEST YOUR CONCEPTS

1. Classify each of the following statements as true or false:
   (a) An immediate operand exists in an instruction itself.
   (b) An instruction cannot change the value of a literal.
   (c) A literal is not entered in the current literal pool if a matching literal already exists in the pool.

(d) The ORIGIN statement indicates what address the next instruction in the program should have.

(e) Processing of the EQU statement results in a change in the address contained in the location counter.

(f) A program containing forward references cannot be assembled in a single pass.

(g) In the Intel 8088 architecture, an operand must exist in the data segment.

(h) The ASSUME directive of Intel assembly language loads a value in a segment register.

EXERCISE 3

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.

3. 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. 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.

5. Develop complete program specifications for the passes of a two-pass assembler indicating the following items:
   (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.
6. Recommend appropriate organizations for the tables and files used in the two-pass assembler of Problem 5.

BIBLIOGRAPHY

Many books discuss assembly language programming and computer architecture. They include Carthy (1996), Hyde (2001), Irvine (2002), Britton (2003), and Detmer (2006). Barron (1969), Donovan (1972) and Beck (1997) are some of the books that cover design of assemblers.

1. Barron, D. W. (1969): *Assemblers and Loaders*, Macdonald Elsevier, London.
2. Beck, L. L. (1997): *System Software: An Introduction to System Programming*, third edition, Addison-Wesley.
3. Britton, R. (2003): *MIPS Assembly Language Programming*, Prentice-Hall, Englewood Cliffs.
4. Carthy, J. (1996): *An Introduction to Assembly Language Programming and Computer Architecture*, Wadsworth Publishing.
5. Calingaert, P. (1979): *Assemblers, Compilers and Program Translation*, Computer Science Press, Maryland.
6. Detmer, R. C. (2006): *Introduction to 80X86 Assembly Language and Computer Architecture*, Jones and Bartlett.
7. Donovan, J. J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
8. Flores, I. (1971): *Assemblers and BAL*, Prentice-Hall, Englewood Cliffs.
9. Hyde, R. (2001): *The Art of Assembly Language*, No Starch Press.
10. Irvine, K. R. (2002): *Assembly Language for Intel-Based Computers*, fourth edition, Pearson Education.

CHAPTER 4

Macros and Macro Preprocessors

In Section 2.8, we discussed why a general purpose program generator should provide features for defining new operations and data. A *macro* is such a feature provided in a programming language. A *macro definition* in a program defines either a new operation or a new method of declaring data. A *macro call* in the program is an invocation of the new operation or the new method of declaring data defined in the macro. It leads to a program generation activity during which the macro call is replaced by a sequence of statements in the programming language. This process is called *macro expansion*.

A macro definition may be written using *formal parameters*. A macro call on such a macro specifies *actual parameters*. Macro expansion is performed by using two kinds of actions. A *lexical substitution* is performed in a statement appearing in a macro definition to replace an occurrence of a formal parameter of the macro by the corresponding actual parameter in a macro call. It results in generation of a statement as a result of the macro call. *Semantic expansion* generates a sequence of statements that is tailored to specific requirements of each call on a macro. It is achieved through *conditional expansion* of statements appearing in a macro definition, whereby only some of the statements in a macro definition are used in expansion of a macro call, and through the use of *expansion time loops*, whereby some of the statements in a macro definition are used several times during expansion of a macro call.

We discuss macros provided in assembly languages. Such macros are handled in two ways. A *macro assembler* performs expansion of each macro call into a sequence of assembly statements and also assembles the resulting assembly program. A *macro preprocessor* merely performs expansion of macro calls and produces an assembly program. In this chapter we discuss the writing of macro definitions, expansion of macro calls and the design of a macro preprocessor.

4.1 INTRODUCTION

A *macro* is a facility for extending a programming language. A macro defines either a new operation or a new method of declaring data in a programming language. The language processor of the language replaces a call on a macro by a sequence of statements that implements the defined operation or the method of declaring data. The specification gap in the writing of a program is reduced if it uses macros to define operations and data that are specific to its application domain. Thus use of macros helps in improving reliability of a program.

Many languages provide facilities for writing macros. Well known examples of these are the programming languages PL/I, C, Ada and C++. Assembly languages of most computer systems also provide such facilities. Where a language does not support 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.

A *macro definition* in a program consists of the name of the macro, a set of *formal parameters*, and a body of code that defines a new operation or a new method of declaring data. Statements in the body of code may use the formal parameters. Use of the macro's name in the mnemonic field of an assembly statement constitutes a *macro call*. The operand field of the macro call statement indicates the *actual parameters* of the call. The language processor replaces a macro call statement by a sequence of assembly language statements that are generated by using the body of code in the macro definition. This process is called *macro expansion*. Two kinds of macro expansion are as follows:

- *Lexical substitution* implies replacement of a character string by another character string during program generation. Lexical substitution is typically employed to replace occurrences of formal parameters by corresponding actual parameters.
- *Semantic expansion* implies generation of statements that are tailored to the requirements of a specific macro call. It makes a macro adaptive, which has the benefits described earlier in Section 1.4.7. For example, if a macro that uses semantic expansion is called using different actual parameters, expansion of the calls may lead to codes which differ in the number, sequence and opcodes of statements. Example 4.1 illustrates how it may be beneficial.

Example 4.1 (Benefits of semantic expansion) The following sequence of statements is used to increment the value stored in a memory word by a given constant:

1. Move the value from the memory word into a CPU register.
2. Increment the value in the CPU register.
3. Move the new value into the memory word.

If a program needs to increment many values stored in memory, it would be useful to define a new operation named `INCR` through an appropriate macro definition and place

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

where *<macro name>* appears in the mnemonic field of an assembly statement and the formal parameter specification appears in the operand field of the statement. A *<formal parameter specification>* is of the form

$$\&<parameter\ name>[<parameter\ kind>] \tag{4.1}$$

A parameter can be either a *positional parameter* or a *keyword parameter*. A parameter is assumed to be a positional parameter by default, i.e., if the specification *<parameter kind>* is omitted. Different kinds of parameters are handled differently during macro expansion; we discuss this issue in Section 4.3.

A macro call has the syntax

$$<macro\ name> \quad [<actual\ parameter\ specification>[,..]] \tag{4.2}$$

where *<macro name>* appears in the mnemonic opcode field of an assembly statement. Actual parameters appear in the operand field of the statement. *<actual parameter specification>* resembles *<operand_specification>* in an assembly language statement (see Chapter 3).

Example 4.2 (Macro definition and call) Figure 4.1 shows the definition of the macro `INCR` that was discussed earlier in Example 4.1. `MACRO` and `MEND` are the macro header and macro end statements, respectively. The prototype statement indicates that `INCR` has three parameters called `MEM_VAL`, `INCR_VAL` and `REG`. Since parameter kind is not specified for any of the parameters, each parameter is assumed to be a positional parameter. Statements with the operation codes `MOVER`, `ADD` and `MOVEM` are model statements. No preprocessor statements are used in this macro. A statement `INCR A, B, AREG` appearing in the program would be considered to be a call on macro `INCR`.

```
MACRO
INCR        &MEM_VAL, &INCR_VAL, &REG
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Figure 4.1 A macro definition

4.3 MACRO EXPANSION

Macro expansion can be performed by using two kinds of language processors. A *macro assembler* performs expansion of each macro call in a program into a sequence of assembly statements and also assembles the resulting assembly program. A *macro preprocessor* merely performs expansion of macro calls in a program. It produces an assembly program in which a macro call has been replaced by statements that resulted from its expansion but statements that were not macro calls have been

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Specifying default values of parameters

If a parameter has the same value in most calls on a macro, this value can be specified as its *default value* in the macro definition itself. If a macro call does not explicitly specify the value of the parameter, the preprocessor uses its default value; otherwise, it uses the value specified in the macro call. This way, a programmer would have to specify a value of the parameter only when it differs from its default value specified in the macro definition.

Default values of keyword parameters can be specified by extending the syntax of formal parameter specification (syntax (4.1)) as follows:

$$\&<\textit{parameter name}>[<\textit{parameter kind}>[<\textit{default value}>]] \qquad (4.3)$$

Example 4.5 (Specifying default values of keyword parameters) If a program uses register AREG for performing most of its arithmetic, most calls on macro INCR_M of Figure 4.2 would contain the specification &REG=AREG. Figure 4.3 shows macro INCR_D, which is analogous to macro INCR_M except that it specifies a default value for parameter REG. Among 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 Example 4.4 because the default value REG would be used during their expansion. The value BREG that is explicitly specified for REG in the third call overrides it default value, so 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
```

Figure 4.3 A macro definition specifying a default value for a keyword parameter

Macros with mixed parameter lists

A macro definition may use both positional and keyword parameters. In such a case, all positional parameters must precede all keyword parameters in a macro call. For example, in the macro call

```
SUMUP       A,B,G=20,H=X
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example 4.8 (Attributes of formal parameters)

```
                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'.

Expansion time variables

An *expansion time variable* (EV) is a variable that is meant for use only during expansion of macro calls. Accordingly, its value can be used only within a macro definition—in a preprocessor statement that assigns a value to an expression variable, in a model statement, and in the expression of an AIF statement. A macro definition must contain the declaration of every expansion time variable that it uses.

Two kinds of expansion time variables exist. A *local expansion time variable* can be used only within one macro definition and it does not retain its value across calls on that macro. A *global expansion time variable* can be used in every macro definition that has a declaration for it and it retains its value across macro calls. Local and global expression variables are created through declaration statements with the following syntax:

```
LCL   <EV specification>[,<EV specification> .. ]
GBL   <EV specification>[,<EV specification> .. ]
```

where *<EV specification>* has the syntax &*<EV name>*, where *<EV name>* is an ordinary string.

Values of expansion time variables can be manipulated through the preprocessor statement SET. It has the following syntax:

```
<EV specification>   SET   <SET-expression>
```

where *<EV specification>* appears in the label field and SET in the opcode field. A SET statement assigns the value of *<SET-expression>* to the expansion time variable specified in *<EV specification>*. Example 4.9 illustrates use of expansion time variables.

Example 4.9 (Expansion time variables)

```
                MACRO
                CONSTANTS
                LCL          &A
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The IRP statement

```
IRP        <formal parameter>, <argument list>
```

The formal parameter mentioned in the IRP statement takes successive values from *<argument list>*. For each value, the statements between the IRP statement and the first ENDM statement following it are expanded once.

Example 4.14 (Macro definition using the IRP statement)

```
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 as follows: Formal parameters &M and &N have the values 4 and 10, respectively. The *<argument list>* in the IRP statement contains three arguments—&M, 7, and &N. Hence the DC statement is visited three times with &Z having the values 4, 7, and 10, respectively.

4.5.3 Semantic Expansion

Semantic expansion is the generation of statements tailored to the requirements of a specific usage. Its use makes a macro adaptive (see Section 1.4.7). Semantic expansion can be achieved by a combination of advanced macro facilities like the AIF, AGO statements and expansion time variables. The CLEAR macro of Example 4.12 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 4.10 is another instance of conditional expansion wherein one of two alternative code sequences is generated depending on peculiarities of actual parameters of a macro call. Example 4.15 illustrates semantic expansion by using the type attribute.

Example 4.15 (Semantic expansion by using the type attribute) Macro CREATE_CONST creates a constant whose value is 25, whose name is given by the second parameter in a call and whose type matches the type of the first parameter.

```
            MACRO
            CREATE_CONST   &X, &Y
            AIF            (T'&X EQ B) .BYTE
&Y          DW             25
            AGO            .OVER
.BYTE       ANOP
&Y          DB             25
.OVER       MEND
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Thus, storing the intermediate code of statements in the MDT would eliminate searches in the tables. An interesting offshoot of this decision is that the first component of the pairs stored in the APT, which was the name of a parameter, is no longer used during macro expansion. Hence instead of using the APT, which contained pairs of the form (<*formal parameter name*>, <*value*>), we can use another table called APTAB which contains only values of parameters. As discussed earlier in the context of parameter named `ABC`, information in the APTAB would be accessed by using the parameter number found in the intermediate code of a statement.

For converting the statement `MOVER AREG, &ABC` into the intermediate code `MOVER AREG, (P,5)`, ordinal numbers have to be assigned to all parameters of a macro. A table named *parameter name table* (PNTAB) could be used for this purpose. Parameter names would be entered in PNTAB in the same order in which they appear in the prototype statement of the macro. The entry # of a parameter's entry in PNTAB would now be its ordinal number. It would be used in the intermediate code of a statement.

In effect, the information (<*formal parameter name*>, <*value*>) in the APT has been split into two tables:

- PNTAB contains formal parameter names.
- APTAB contains values of formal parameters, most of which are actual parameters and others are defaults.

Note that the PNTAB is used while processing a macro definition while the APTAB is used during macro expansion.

Similar analysis leads to splitting of the execution time variables' table (EVT) into EVNTAB and EVTAB and splitting of the sequencing symbol table (SST) into SSNTAB and SSTAB. The name of an expansion time variable is entered in the EVNTAB while processing its declaration. The name of a sequencing symbol is entered in the SSNTAB while processing the definition of the sequencing symbol or a reference to it, whichever occurs earlier. This aspect resembles construction of the symbol table in a single-pass assembler (see Chapter 3).

The *parameter default table* (PDT) can be split analogously; however, some more simplifications are possible. The positional parameters (if any) of a macro appear before keyword parameters in the prototype statement. Hence if a macro BETA has p positional parameters and k keyword parameters, the keyword parameters have the ordinal numbers $p+1 \ldots p+k$. Due to this numbering, the following two kinds of redundancies appear in the PDT: The first component of each entry is redundant as in the APTAB and the 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 have only k entries in it. To note that the first entry

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

It invokes Algorithm 4.2 for every macro definition in the program. Step 1 initializes pointers to the *sequencing symbol name table* (SSNTAB) and *parameter name table* (PNTAB). Step 2 processes the prototype statement to collect names of parameters and their defaults in the *parameter name table* (PNTAB) and the *keyword parameter default table* (KPDTAB), respectively, and sets the fields *name*, *#PP* and *#KP* of the *macro name table* (MNT) entry. Step 3 generates the intermediate code for model statements and the preprocessor statements SET, AIF, and AGO appearing in the body of code in macro definition. This step also enters names of local expansion time variables in the EVNTAB. Note that an LCL statement is not entered in the MDT. The algorithm does not handle GBL statements; their handling is left as an exercise to the reader.

Algorithm 4.2 (Processing of a macro definition)

1. *PNTAB_ptr* := 1;
   SSNTAB_ptr := 1;
2. Process the macro prototype statement and form the MNT entry for the macro
   (a) *name* := macro name; #PP := 0; #KP := 0;
   (b) For each positional parameter
      (i) Enter *parameter name* in PNTAB [*PNTAB_ptr*].
      (ii) *PNTAB_ptr* := *PNTAB_ptr* + 1;
      (iii) #PP := #PP + 1;
   (c) KPDTP := *KPDTAB_ptr*;
   (d) For each keyword parameter
      (i) Enter *parameter name* and *default value* (if any), in the entry 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
      For each expansion time variable declared in the statement:
      Enter name of the expansion time variable in EVNTAB.
      #EV := #EV + 1;
   (b) If a model statement then

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In this code macro calls appearing in the source program have been expanded but the expanded statements may themselves contain macro calls. The macro expansion scheme can be applied to the first level expanded code to expand these macro calls and so on, until we obtain a program form which does not contain any macro calls. This scheme would be slow because it requires a number of passes of macro expansion.

A more efficient approach would be to examine each statement generated during macro expansion to check whether it is itself a macro call. If so, a provision can be made to expand this call before continuing with the expansion of its parent macro call. This approach avoids multiple passes of macro expansion, thus ensuring processing efficiency. The macro expansion scheme of Section 4.6.4 would have to be modified to suit this approach.

Consider the situation during generation of the ADD statement marked [3] in Figure 4.5. Expansion of two macro calls would be in progress at this moment—the outer macro COMPUTE and the inner macro INCR_D. The model statements of INCR_D would be expanded using the expansion time data structures MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr*. A MEND statement encountered during the expansion would signal completion of the inner macro's expansion. It should lead to resumption of the outer macro's expansion, so the MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr* would have to be restored to the values they had while the macro COMPUTE was being expanded. When the MEND statement is encountered during the processing of COMPUTE, it would signal that expansion of the nested macro call is complete.

Thus, the following two provisions are needed to implement the expansion of nested macro calls:

1. Each macro under expansion should have its own set of the data structures MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr*.
2. An *expansion nesting counter* (*Nest_cntr*) should be maintained to count the number of nested macro calls. *Nest_cntr* would be incremented when a macro call is recognized and decremented when a MEND statement is encountered. Thus *Nest_cntr* = 1 would indicate that a first level macro call is being expanded, while *Nest_cntr* > 1 would indicate that a nested macro call is being expanded.

The first provision implies creation of 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 would provide 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 copies may never be used. If too few are created, processing of an assembly program would have

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

MEC, *EVTAB_ptr*, APTAB and EVTAB are allocated on the stack in that order. As explained below, the *APTAB_ptr* is not needed. During macro expansion, the various data structures are accessed with reference to the value contained in RB as follows:

Data structure	*Address*
Reserved pointer	0(RB)
MEC	1(RB)
EVTAB_ptr	2(RB)
APTAB	3(RB) to #e_{APTAB}+2(RB)
EVTAB	#e_{APTAB}+3(RB) to #e_{APTAB}+ #e_{EVTAB}+2(RB)

where 1(RB) stands for 'contents of RB+1' and #e_{APTAB}, #e_{EVTAB} are the number of entries in the APTAB and EVTAB, respectively. Note that the first entry of APTAB always has the address 3(RB). It eliminates the need for *APTAB_ptr*.

At a MEND statement, a record would be popped off the stack by setting TOS to the end of the previous record. It would now be necessary to set RB to point to the start of the previous record in stack. It 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. It has the effect of restoring access to the expansion time data structures used by the outer macro.

Actions at the start of macro expansion are summarized in Table 4.1. 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 the first word of the new expansion record. 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.

Table 4.1 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;

Actions at the end of expansion are summarized in Table 4.2. 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

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The use of the macro's name in the program constitutes a *macro call*. The macro call statement contains *actual parameters* of the call. The language processor replaces the macro call statement by a sequence of statements that implements the defined operation or the method of declaring data. This action is called *macro expansion*. Macro expansion is performed by considering statements appearing in the macro's definition and actual parameters used in a macro call. In *lexical substitution* the language processor merely substitutes actual parameters in place of formal parameters used in a statement. In *semantic expansion*, it generates a sequence of statements that is tailored to the requirements of each call. We discuss expansion of macros in an assembly language program by using a *macro preprocessor*, which performs macro expansion and produces an assembly language program. This program can be assembled by using a conventional assembler.

A macro's definition consists of a *macro prototype* statement, which mentions the name and parameters of the macro, *model statements* that are used in macro expansion, and *macro preprocessor statements* that facilitate semantic expansion. During macro expansion, the preprocessor uses a macro expansion counter (MEC) to keep track of which statement in the macro's body should be considered next for expansion. During expansion of that statement, it replaces a formal parameter by its value. The formal parameters of a macro can be of two kinds. A *positional parameter* is one whose value is the actual parameter that occupies the same ordinal position in the actual parameter list. In the case of a *keyword parameter*, its value is explicitly indicated in the actual parameter list by using its name. A macro prototype statement can also indicate default values for keyword parameters. Semantic expansion is achieved by altering the expansion time control flow through the use of macro preprocessor statements to achieve either *conditional expansion* or *expansion time loops*. The AIF and AGO statements perform conditional and unconditional transfer of expansion time control flow. A macro can use *expansion time variables* to facilitate semantic expansion.

Algorithms for use in a macro preprocessor are discussed. An intermediate code is generated from a macro's definition to facilitate its expansion. The macro preprocessor maintains tables to store information about values of positional and keyword parameters, and the macro expansion counter to know which model statement should be expanded next. Macro calls may be nested. When a nested call is encountered, the preprocessor suspends the macro expansion in which it was engaged and expands the inner macro call before resuming the suspended expansion. A stack is used to hold the macro expansion data structures during processing of nested macro calls.

TEST YOUR CONCEPTS

1. Classify each of the following statements as true or false:
   (a) Semantic expansion makes a macro adaptive.
   (b) A macro preprocessor expands macro calls in an assembly program and also assembles the expanded program.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(a) Code space requirements
(b) Execution speed
(c) Processing required by the assembler
(d) Flexibility and generality.

10. An assembly language program performs a certain action at 10 places. Under what conditions would you code this action as
    (a) A macro?
    (b) A subroutine?

    Justify your answer with the help of appropriate examples.
11. Solve the first few problems of this exercise by using REPT and IRP statements.
12. Extend the macro preprocessor described in this chapter to support the following features:
    (a) REPT and IRP statements discussed in Section 4.5.2
    (b) Global expansion time variables
    (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.

CHAPTER 5

Linkers and Loaders

A programming language provides a library of routines for tasks such as creation, reading and writing of files; and evaluation of mathematical and other functions provided in the language. While compiling a program involving any of these tasks, the language translator generates a call on the appropriate library routine. Thus, the code of a target program cannot execute all by itself even if it is in the machine language; it has to be combined with codes of the library routines before it can be executed. A program may also wish to invoke other programs written in the programming language. In such cases, its code has to be similarly combined with codes of these programs.

The *linker* is a system program that combines the code of a target program with codes of other programs and library routines. To facilitate linking, the language translator builds an *object module* for a program which contains both target code of the program and information about other programs and library routines that it needs to invoke during its execution. The linker extracts this information from the object module, locates the needed programs and routines and combines them with the target code of the program to produce a program in the machine language that can execute without requiring the assistance of any other program. Such a program is called a *binary program*. The *loader* is a system program that loads a binary program in memory for execution.

In this chapter we discuss the format of object modules, the functions of *linking*, *relocation* and *loading*; different ways of performing these functions and their comparative benefits.

5.1 INTRODUCTION

Execution of a program written in a programming language is achieved in the following four steps:

- *Translation:* A program is translated into a target program.
- *Linking:* The code of a target program is combined with codes of those programs and library routines that it calls.
- *Relocation:* A program may have been coded or translated with the idea of executing it in a specific area of memory. However, the operating system may have used that memory area for another purpose, so it may allocate a different memory area for the program's execution. *Relocation* is the action of changing the memory addresses used in the code of the program so that it can execute correctly in the allocated memory area.
- *Loading:* The program is loaded in a specific memory area for execution.

Figure 5.1 contains a schematic showing these steps in the execution of a program. It uses many system programs. The *translator* generates a program form called the *object module* for the program. The *linker* program performs linking and relocation of a set of object modules to produce a ready-to-execute program form called the *binary program*. The *loader* program loads a binary program in memory for execution. It may perform relocation during loading. As shown in the schematic, the object module(s) and binary programs can be stored in files so that they can be used repeatedly.

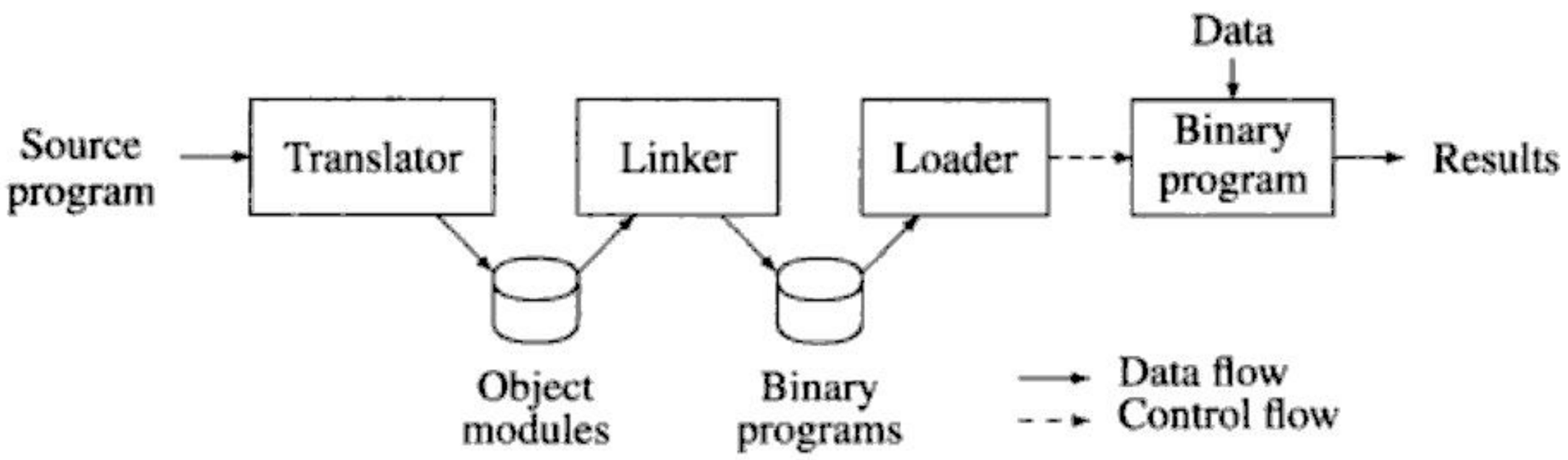


Figure 5.1 Schematic of a program's execution

Translated, linked and load time addresses

While compiling a program, the language translator needs to know what memory address the first memory word of the target program should have. We call it the *origin* of the program. The origin should be either specified to the language translator, or it would have to be assumed. (In an assembly program, the origin is specified in a `START` or `ORIGIN` statement.) The origin may be changed before the program reaches

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Let IRR_P be the set of instructions in program P that require relocation. Following (5.2), relocation of program P can be performed by computing the relocation factor for P and adding it to the translation time address(es) used in every instruction *i* included in IRR_P.

Example 5.3 (Relocation of a program) For the program of Example 5.2

$$\begin{aligned}\text{Relocation factor} &= 900 - 500\\ &= 400.\end{aligned}$$

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. The instruction with translated address 538 contains the address 501 in the operand field. Adding 400 to this address makes it 901. It implements the relocation explained in Example 5.2.

5.2.2 Linking

A *program unit* is any program or routine that is to be linked with another program or routine. For simplicity, we assume that each program unit has been assembled separately to produce an object module (see Figure 5.1).

Let an application consist of a set of program units SP = $\{P_i\}$. Now consider a program unit P_i that requires the use of another program unit P_j during its execution—either it uses the address of some instruction in P_j in one of its instructions, possibly in a subroutine call instruction, or it uses the address of some data defined in P_j in one of its instructions. To form a binary program by combining P_i and P_j, linked addresses of relevant instructions or data located in P_j have to be supplied to P_i's instructions. It is achieved by using the following linking related concepts:

- *Public definition:* A symbol defined in a program unit that may be referenced in other program units.
- *External reference:* A reference to a symbol that is not defined in the program unit containing the reference.

Thus if other program units wish to use an instruction or data of program unit P_j, a symbol should be associated with that instruction or data and the symbol should be declared as a public definition in P_j. A use of the symbol in program unit P_i would constitute an external reference.

EXTRN and ENTRY statements

An `ENTRY` statement in a program unit lists the public definitions of the program unit. An `EXTRN` statement lists the symbols to which external references are made in the program unit. Example 5.4 illustrates use of these statements.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

5.3.1 Scheme for Relocation

The linker uses an area of memory called the *work area* for constructing the binary program. It loads the machine language program found in the *program* component of an object module into the work area and relocates the address sensitive instructions in it by processing entries of the RELOCTAB. For each RELOCTAB entry, the linker determines the address of the word in the *work area* that contains the address sensitive instruction and relocates it. The details of the address computation would depend on whether the linker loads and relocates one object module at a time, or loads all object modules that are to be linked together into the *work area* before performing relocation. The former approach is assumed in this section because it requires a small *work area*---the work area needs to be only as large as the largest of the programs being linked.

Algorithm 5.1 (Program relocation)

1. *program_linked_origin* := <*link origin*> from the `linker` command;
2. For each object module mentioned in the `linker` command
   (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 contained in the *program* component of the object module into the *work_area*.
   (d) Read RELOCTAB of the object module.
   (e) For each entry in RELOCTAB
      (i) *translated_address* := address found 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 found in the word that has 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 5.2.1. *program_linked_origin* contains the linked address that should be assigned to an object module. It is initialized to <*link origin*> from the `linker` command and after processing an object module it is incremented by the size of the object module in Step 2(f) so that the next object module would be granted the next available linked address. For each entry in the RELOCTAB, Step 2(e)(ii) computes the address of the word in the *work area* that contains the address sensitive instruction. It is computed by calculating the offset of this instruction within the program of the object module and adding it to the start address of the *work area*.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the relocation factor and perform relocation. It is achieved as follows: The computer provides a 'branch to subroutine' instruction. When this instruction is executed, the CPU would transfer control to the first instruction of a subroutine and load a *return address* in a CPU register. The return address is simply the address of the instruction that follows the 'branch to subroutine' instruction. The code of the subroutine would use the return address to transfer control back to the program that had invoked it. The relocating logic in a self-relocating program would execute a 'branch to subroutine' instruction that would merely transfer control to its own next instruction. This way the relocating logic would retain control of the execution, but the CPU would load the address of its next instruction into a CPU register as the return address. The relocating logic can obtain its own load address by subtracting the length of the 'branch to subroutine' instruction from this return address, and use it to perform relocation of the program. After relocation it would transfer control to the instructions that implement the program's logic.

5.5 LINKING IN 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 schemes of linking and relocation developed in the previous section. The Intel 8088 uses segment-based addressing which was discussed in Section 3.5. We begin by discussing the relocation and linking requirements in segment-based addressing.

5.5.1 Relocation and Linking Requirements in Segment-Based Addressing

In segment-based addressing, a memory address has two components—the start address of a segment and an offset within a segment. To address a memory operand, an instruction mentions its offset and a segment register that is expected to contain the segment's address. This method of addressing avoids use of absolute addresses of memory operands in instructions, so instructions are not address sensitive. Example 5.9 explains how use of segment-based addressing reduces the relocation requirements of a program.

Example 5.9 (Relocation and linking in segment-based addressing) The program of Figure 5.4 is written in the assembly language of the Intel 8088, which was discussed in Section 3.5.3. The `ASSUME` statement indicates that the segment registers CS and DS would contain addresses of the segments `SAMPLE` and `DATA_HERE` during the program's execution. Hence all memory addressing is performed by using suitable displacements from contents of the CS and DS registers. The program itself loads an address in the DS register, whereas the CS register is presumably loaded either by a calling program or by the OS. The translation time address of symbol `A` is 0196. In statement 16, a reference to `A` is assembled as a displacement of 196 from the contents of the CS register. If segment `SAMPLE` is to be loaded in memory starting at the address 2000, 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 would be the correct address 2196. A similar situation exists with the reference to `B`

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 5.1 Object records of Intel 8088

Record type	*Id (Hex)*	*Description*
THEADR	80	Translator header record
LNAMES	96	List of names record
SEGDEF	99	Segment definition record
EXTDEF	8C	External names definition record
PUBDEF	91	Public names definition record
LEDATA	A1	Enumerated data (binary image)
LIDATA	A3	Repeated data (binary image)
FIXUPP	9D	Fixup (i.e., relocation) record
MODEND	8B	Module end record

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 simply contains a list of names. A SEGDEF record indicates the name of a segment by using an index into the list of names contained in LNAMES records. 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, and the alignment requirement of its base address (whether byte, word or paragraph, i.e., 16 byte, alignment). The *attributes* field also contains the origin specification for an absolute segment. Stack segments having the same name are concatenated with each other. Common segments are used to implement the COMMON statement of Fortran. Hence COMMON segments having the same name are overlapped with one another.

EXTDEF and PUBDEF records

The EXTDEF record contains a list of symbolic names to which external references are made in the segments of this module. A FIXUPP record refers to an external symbolic name by using an index into this list. A PUBDEF record contains a list of public names that are declared in a segment of the object module. The segment is identified by the *base specification* field. Each (*name*, *offset*) pair in the record defines one public name by specifying the symbolic name and its offset within the segment designated by the base specification.

LEDATA records

An LEDATA record contains the binary image of the code and data 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.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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	···	8C02 02 01 01 00 20H	···	FIXUPP
A0H	···	01 0279 A1 00 00	···	LEDATA
9CH	···	8801 06 01 03	···	FIXUPP
8AH	···	80H	···	MODEND

Figure 5.7 MS DOS object modules

```
        LEA         BX, ALPHA+20H
```

This statement is assumed to have been assembled with zeroes in the operand field. It is fixed by using the code 8C in *locat*, which implies *loc code* = '3', code '2' in *fix data*, and putting the displacement 20H in the *target displacement* field of the FIXUPP record. (Note that code '6' could have been used in the *fix data* field as discussed in Example 5.10).

5.5.3 Design of the Linker

We shall design a program named LINKER which performs both linking and relocation of absolute segments and of relocatable segments that cannot be combined with other relocatable segments. Its output is a binary program which resembles a program with .COM extension in MS DOS. This program is not relocated by the loader

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The LINKER performs autolinking when it finds that some external name `alpha` referenced in one of the object modules is not present in the NTAB. An object module that contains a public definition for `alpha` is located in one of the library files in <*list of library files*> and it is included in the set of object modules to be linked. To enable resolution of the external reference to `alpha`, the public definitions located in the new object module should be added to the NTAB. It is implemented by performing the first pass of the LINKER program on the new object module. Processing of the FIXUPP record which triggered off autolinking for `alpha` is then resumed.

Algorithm 5.4 (Second pass of the LINKER program)

1. *list_of_object_modules* := Object modules named in the 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) *i* := *name index*;
       (ii) *segment_name* := NAMELIST [*i*];
       (iii) Enter (*segment_name*, *linked address* from NTAB) in the i^{th} entry of the SEGTAB.
   (c) If an EXTDEF record
       (i) *external_name* := name from EXTDEF record;
       (ii) If *external_name* is not found in NTAB, then
            A. Locate an object module in the library which contains *external_name* as a segment name or a public definition.
            B. Add name of the object module to *list_of_object_modules*.
            C. Add the symbols defined as public definitions in the new module to the NTAB by performing the first pass, i.e., Algorithm 5.3, for the new object module.
       (iii) Enter (*external_name*, *linked address* from NTAB) in EXTTAB.
   (d) If an LEDATA record
       (i) *i* := *segment index*; *d* := *data offset*;
       (ii) *program_linked_origin* := SEGTAB [i].*linked address*;
       (iii) *address_in_work_area* := address of *work_area* + *program_linked_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

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

produces a single binary program containing all overlays and stores it in <*executable file*>.

The binary program produced by the MS DOS linker contains two provisions to support overlays. First, an overlay manager is included in the executable file. It is responsible for loading overlays when needed. Second, each procedure call that crosses an overlay boundary is replaced by an interrupt producing instruction. This interrupt would be processed by the overlay manager and the appropriate overlay would be loaded in memory. If each overlay was structured into a separate binary program, a procedure call which crosses an overlay boundary would lead to an interrupt which would be attended to by the kernel of the operating system. The kernel would transfer control to the loader to load the appropriate binary program in memory. This way an overlay manager need not be made a part of an overlay structured program.

Changes in the LINKER algorithms

The basic change required in the LINKER algorithms of Section 5.5.3 is in the assignment of linked addresses to segments. The variable *program_linked_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_linked_origin* should 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. The LINKER has to identify an inter-overlay call and determine the destination overlay. This information should be made available to the overlay manager or the kernel of the OS which is activated through the interrupt instruction. Handling of interrupts is discussed later in Section 10.2.4.

An open issue in the linking of overlay structured programs is the handling of object modules that would be added through autolinking: Should these object modules be added to the current overlay or to the root of the program? The latter approach would be appropriate if an autolinked procedure uses *static* or *own* data, however it may increase the memory requirement of the program.

5.7 DYNAMIC LINKING

In *static linking*, the linker links all modules of a program before its execution begins; it produces a binary program that does not contain any unresolved external references. The linking schemes discussed in previous sections have been static linking schemes. If several statically linked programs use the same module from a library, each program will get a private copy of the module. If many programs that use the module are in execution at the same time, many copies of the module might be present in memory.

Dynamic linking is performed during execution of a binary program. The linker is invoked when an unresolved external reference is encountered during its execution. The linker resolves the external reference and resumes execution of the program.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The two key issues that need to be addressed by a linker are *relocation* of a program, which enables a program to execute in an area of memory other than the memory area for which it was translated, and *linking* of a program with other programs and library routines. An *address sensitive instruction* in a program is an instruction that contains an absolute memory address. Such an instruction cannot execute correctly if the program is moved to another area of memory. Relocation involves appropriate modification of the address sensitive instructions in a program so that the program can execute correctly in a different area of memory. It is achieved as follows: The *translated origin* of a program is the address assigned to its first instruction by the translator, and the *linked origin* is the address assigned to its first instruction by the linker. The *relocation factor* is the difference between these two addresses. The linker modifies the address used in an address-sensitive instruction by adding the relocation factor to it.

A language translator produces a program form called *object module* which contains information useful for linking. A *public definition* is a symbol defined in the program that may be referred to in other programs. An *external reference* in a program is a reference in one of its instructions to a symbol that is defined in some other program. To perform linking, the linker processes all the object modules that are to be linked together, performs memory allocation to assign linked addresses to all public definitions, and replaces each external reference by the linked address of the referenced symbol. These tasks can be performed by using a classic two-pass organization.

Linking can be performed either statically or dynamically. Schemes for both kinds of linking are discussed. Design of a linker using the MS DOS object modules is developed. Design of a loader and its variants called *absolute loaders*, *relocating loaders* and *bootstrap loaders* are also discussed.

TEST YOUR CONCEPTS

1. Classify each of the following statements as true or false:
   (a) A program must have the same load origin every time it is executed.
   (b) Every symbol defined in a program constitutes a public definition.
   (c) Relocation factor cannot be negative.
   (d) While linking many object modules to form a binary program, the linker assigns the same relocation factor to all object modules.
   (e) In a two-pass linker, external references are resolved in the first pass.
   (f) A dynamic linker resolves external references during a program's execution.
   (g) A self-relocating program is loaded in memory by using an absolute loader.
   (h) Use of a segment register reduces the number of address sensitive instructions in a program.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

12. Modify the passes of LINKER to perform the linking of an overlay structured program. (*Hint:* Should each overlay have a separate NTAB?)
13. This chapter has discussed *static relocation*. *Dynamic relocation* implies relocation during the execution of a program, for example, 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.
14. 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.
15. Compare the .COM files in MS DOS with an object module. (MS DOS object modules are contained in files with .OBJ extension.)
16. 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. Levine, J. (1999): *Linkers and Loaders*, Elsevier.
3. Presser, L. and J. R. White (1972): "Linkers and Loaders," *Computing Surveys*, 4 (3).
4. Wilder, W. L. (1980): "Comparing load and go and link/load compiler organizations," *Proc. AFIPS NCC*, 49, 823–825.

CHAPTER 6

Scanning and Parsing

We saw two kinds of language processors in Chapter 2. A *program generator* generates a program in a procedure oriented language from its specification. A *translator* helps in implementing a program written in a programming language—a *compiler* generates an equivalent program in a machine language which has to be executed to achieve the computations described in the program, whereas an *interpreter* itself performs the computations described in the program.

The input to a language processor is written in its *source language*. The source language for a program generator is the language used for specifying the program. The source language for a translator is the programming language in which a program is written. As discussed in Section 2.3.2.1, the front end of the language processor performs *lexical analysis*, *syntax analysis* and *semantic analysis* of its input. A *grammar* of the source language forms the basis of these analyses. It is comprised of the rules for forming lexical units such as identifiers and constants, and syntactic constructs such as expressions and statements in the source language. A grammar should provide sufficient flexibility to users of the source language. It should also avoid ambiguity so that the meaning of a program would be uniquely defined.

A *scanner* is that part of the language processor that performs lexical analysis of the input in accordance with the grammar, whereas a *parser* is that part that performs syntax analysis. We discuss various kinds of grammars and their properties and the fundamental techniques of scanning and parsing. We also discuss practical variants of parsing techniques that are both efficient and capable of providing precise diagnostics. In the end, we discuss the language processor development tools LEX and YACC that are used to generate programs that would perform scanning and parsing, respectively.

6.1 PROGRAMMING LANGUAGE GRAMMARS

A *formal language* is one that can be considered to be a collection of valid sentences, where each sentence can be looked upon as a sequence of words, and each word as a sequence of graphic symbols acceptable in the language. A *formal language grammar* is a set of lexical and syntactic rules which precisely specify the words and sentences of a language, respectively.

Programming languages are formal languages. Natural languages are not formal languages because their vocabularies cannot be specified by a set of lexical rules. However, a subset of a natural language may be a formal language; we will use examples from a very limited subset of English.

Terminal symbols, alphabet and strings

The *alphabet* of a language L is the collection of graphic symbols such as letters and punctuation marks used in L. It is denoted by the Greek symbol Σ. 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* of L. The alphabet can be represented by 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 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

$$\{ :, ;, ',', \ldots \}$$

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 represented by the symbol '.'; this symbol is omitted if the concatenation operation is obvious from the context. 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 $\mathtt{a}.\varepsilon \equiv \varepsilon.\mathtt{a} \equiv \mathtt{a}$.

Nonterminal symbols

A *nonterminal symbol* is the name of a syntax category of a language, e.g., noun, verb, etc. A nonterminal symbol is written as a single capital letter, or as a name

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

according to Grammar (6.2) we perform reductions in the following sequence:

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 because

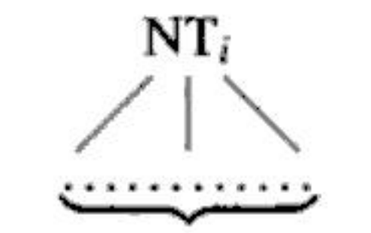
$$\texttt{the boy ate an apple} \xrightarrow{*} < \textit{Sentence} >.$$

Parse trees

The sequence of derivations that produces a string from the distinguished symbol or the sequence of reductions that reduces the string to the distinguished symbol reveals the string's syntactic structure. We use a *parse tree* to depict the syntactic structure of a string. A part of the parse tree is built at every derivation or reduction.

Consider the production S ::= ... NT_i .... When a derivation is made according to this production, we build the following elemental parse tree:

If the next step in the derivation replaces NT_i by some string β, we build the following elemental parse tree to depict this derivation



and combine this tree with the previous tree 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, i.e., by building an elemental parse tree to indicate how a string of terminal and nonterminal symbols is replaced by a single nonterminal. Such a tree would grow in the upward direction. Example 6.6 illustrates parse trees.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

a string containing a single terminal symbol and a single nonterminal symbol—gives some practical advantages in scanning (we shall see this aspect in Chapter 7). 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 $< const >$ and $< id >$ in Grammar (6.3) are in fact Type-3 in nature. It can be seen clearly when we rewrite the production for $< id >$ in a form that resembles B `t` | `t` as

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

where l and d stand for a letter and a 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 nonterminal symbol in the RHS alternative appears at the extreme left or extreme right.

Operator grammars

Definition 6.2 (Operator grammar (OG)) *Productions of an* operator grammar *do not contain two or more consecutive nonterminal symbols in any RHS alternative.*

Thus, nonterminal symbols occurring in an RHS string are separated by one or more terminal symbols. Each such terminal symbol is called an *operator* of the grammar. As discussed later in Chapter 7, strings specified by using an operator grammar can be parsed in a simple and efficient manner. Example 6.7 discusses an operator grammar.

Example 6.7 (Operator grammar) Grammar (6.3) is an operator grammar because it satisfies Definition 6.2. The symbols '↑', '*', '+', '(' and ')' are the operators of the grammar.

6.1.2 Ambiguity in Grammatic Specification

A grammar is ambiguous if a string can be interpreted in two or more ways by using it. In natural languages, ambiguity may concern the meaning of a word, the syntax category of a word, or the syntactic structure of a construct. For example, a word can have multiple meanings or it can be both a noun and a verb (e.g., the word 'base' can mean a chemical base, a military base, or the construction of a foundation), and a sentence can have more than one syntactic structure (e.g., 'police was ordered to stop speeding on roads').

In a formal language grammar, ambiguity would arise if identical strings can occur on the RHS of two or more productions. For example, if a grammar has the

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

transition, or simply a *transition*. Thus, the current state of a finite state automaton is determined by the string of source symbols it has processed so far. If a string is valid, this state is called a *final state* of the finite state automaton. A formal definition of a finite state automaton follows.

Definition 6.3 (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 states in S on encountering symbols in* Σ.

A *deterministic finite state automaton* (DFA) is a finite state automaton none of whose states has two or more transitions for the same source symbol. The DFA has the property that it reaches a unique state for every source string input to it.

Each transition of the DFA can be represented by the triple (old state, source symbol, new state). The transitions in T can thus be represented by a set of such triples. Alternatively, the transitions of the DFA 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 Σ. The entry STT(s_i, *symb*) in the table indicates the id of the new state which the DFA would enter if the source symbol *symb* was input to it while it was in state s_i. Thus, the entry STT(s_i, *symb*) = s_j and the triple (s_i, *symb*, s_j) contain equivalent information. The entry STT(s_i, *symb*) would be blank if the DFA does not contain a transition out of state s_i for *symb*.

A deterministic finite state automaton can be represented pictorially by representing each of its states by a circle and each of its state transitions by an arrow drawn from the old state to the new state and labelled by the source symbol that causes the transition. A final state of the DFA is represented by a double circle.

Operation of a DFA is controlled by its current state, which we call s_c. Its actions are limited to the following: Given a source symbol x at its input, it checks to see whether STT(s_c, x) is defined—that is, whether STT(s_c, x) = s_j, for some s_j. If so, it makes a transition to state s_j; we say that it has *recognized* symbol x. If the entry STT(s_c, x) is blank, it indicates an error and stops.

Example 6.11 (DFA for integer strings) Figure 6.3 shows a DFA to recognize integer strings according to the Type-3 rule

$$<integer> ::= d \mid <integer>\ d$$

where d represents a digit. Part (a) of the figure shows the state transition table for the DFA. Part (b) is a pictorial representation of the DFA. The initial and final states of the DFA are *Start* and *Int* respectively. Transitions during the recognition of string 539 would be as shown in the following table because each of 5, 3 and 9 is a d.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 6.2 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 the string in the table of real constants. Return the token $\boxed{Real\ \#m}$ }
$l(l \mid d)^*$	{Compare the string with reserved words. If a match is found, return the token $\boxed{Kw\ \#k}$; otherwise, enter the string in the symbol table, say, in entry i and return the token $\boxed{Id\ \#i}$ }

6.3 PARSING

As discussed in Section 2.3.2.1, parsing determines the grammatical structure of a sentence. In Section 6.1, we discussed how parsing can be achieved either through the *derivation* of a string from a nonterminal symbol, or through the *reduction* of a string to a nonterminal symbol. It gives rise to two fundamental approaches to parsing called *top-down parsing* and *bottom-up parsing*, respectively. If a string is parsed successfully, the parser should build an intermediate code for it; otherwise, it should issue diagnostic messages reporting the cause and nature of error(s) in the string. The intermediate code is either a *parse tree* or an *abstract syntax tree* for the string; the latter is more common for reasons described below.

Parse tree and abstract syntax trees

A *parse tree* depicts the steps in the parsing of a source string according to a grammar, so it is useful for understanding the process of parsing. However, it is a poor intermediate representation for a source string because much of the information contained in it is not useful for subsequent processing in the compiler. 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 her own purposes. Consequently, a source string may have different abstract syntax tress in different compilers. However, it has a unique parse tree. Example 6.14 illustrates the nature of the information in the parse tree and an abstract syntax tree for an expression.

Example 6.14 (Parse tree and abstract syntax tree) Figure 6.6(a) shows a parse tree for the source string `a+b*c` according to Grammar (6.3). It shows how the identifier `a` is reduced to a primary, factor, term, and finally to an expression. This information is not useful for further processing of the string in the compiler. Figure 6.6(b) shows an abstract syntax tree for the same string. It simply shows that `a` is an operand of the

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

1. *Source string marker* (SSM): At any stage of parsing, the source string marker points to the first unmatched symbol in the source string.
2. *Prediction making mechanism:* This mechanism makes a prediction for the leftmost nonterminal symbol in the CSF. When called upon to make a prediction, it selects the RHS alternatives in a production of the nonterminal symbol in a left-to-right manner. It makes a prediction according to the *next* RHS alternative in a production if a continuation check had failed during parsing according to the previous prediction; otherwise, it makes a prediction according to the first RHS alternative. This way, the parser will be ale to parse any valid string in L_G.
3. *Matching and backtracking mechanism:* It implements the continuation check by using the SSM. If the check succeeds, the SSM would be incremented by the number of symbols that were matched. If the check fails, backtracking would be performed by resetting the CSF and the SSM to appropriate earlier values. A stack can be used to support backtracking. A stack entry would contain information about a derivation. It would contain three fields—a nonterminal symbol, an integer indicating which RHS alternative of the nonterminal symbol was under consideration, and the value of the SSM at the start of the derivation. Operation of the parser would be controlled by information in the TOS entry of this stack. Entries would be popped off the stack when derivations match completely or are rejected.

Continuation check and backtracking is performed in Step 3 of Algorithm 6.1. A complete algorithm for top-down parsing can be found in (Dhamdhere, 1997). Example 6.15 illustrates operation of a top-down parser.

Example 6.15 (Top-down parsing) The string $< id > + < id > * < id >$, which is the lexically analysed version of the source string a+b*c, is to be parsed according to the grammar

$$\begin{array}{lcl} S & ::= & E \\ E & ::= & T + E \mid T \\ T & ::= & V * T \mid V \\ V & ::= & < id > \end{array} \qquad (6.5)$$

The terminal symbols of this grammar are $*$, $+$, and $< id >$. The first few steps in the parse are as follows:

1. SSM := 1; CSF := S; Make a prediction according to the production for S. Hence CSF := E.
2. Make a prediction according to the first RHS alternative of the production for E. It is E ⇒ T + E. Hence CSF becomes T + E.
3. Make a prediction according to the first RHS alternative of the production for T. It is T ⇒ V * T. Hence CSF becomes V * T + E.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example 6.16 (Predictive parsing) Table 6.4 summarizes the parsing of $< id > + < id > * < id >$ according to Grammar (6.8). Note that *Next symbol* is the symbol pointed to by the SSM.

Table 6.4 Top-down parsing without backtracking

Sr. No.	*Current sentential form (CSF)*	*Next 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.	$< id > + < id > * < id >$	–	–

To start with, CSF = E and the next symbol is '$<id>$'. The first three steps are obvious because productions for the three leftmost nonterminals have only one RHS alternative each. In the 4^{th} step, the next symbol is '+' and the leftmost NT is T''. The parser has to decide whether to make the prediction $T'' \Rightarrow * T$ or $T'' \Rightarrow \varepsilon$. Since the next symbol is not *, it makes the prediction $T'' \Rightarrow \varepsilon$.

6.3.1.1 Practical Top-Down Parsing

A recursive descent parser

A *recursive descent parser* employs a variant of top-down parsing without backtracking. It is so named because it uses a set of mutually-recursive procedures to perform parsing. Salient advantages of recursive descent parsing are its simplicity and generality. It can be implemented in any language that supports recursive procedures.

To implement recursive descent parsing, a left-factored grammar is modified to make repeated occurrences of strings more explicit. For example, Grammar (6.8) would be rewritten as follows

$$\begin{aligned} E &::= T\ \{+\ T\}^* \\ T &::= V\ \{*\ V\}^* \\ V &::= < id > \end{aligned} \tag{6.9}$$

where the notation $\{..\}^*$ indicates zero or more occurrences of the enclosed specification. The parser is written such that it has one procedure for each nonterminal

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

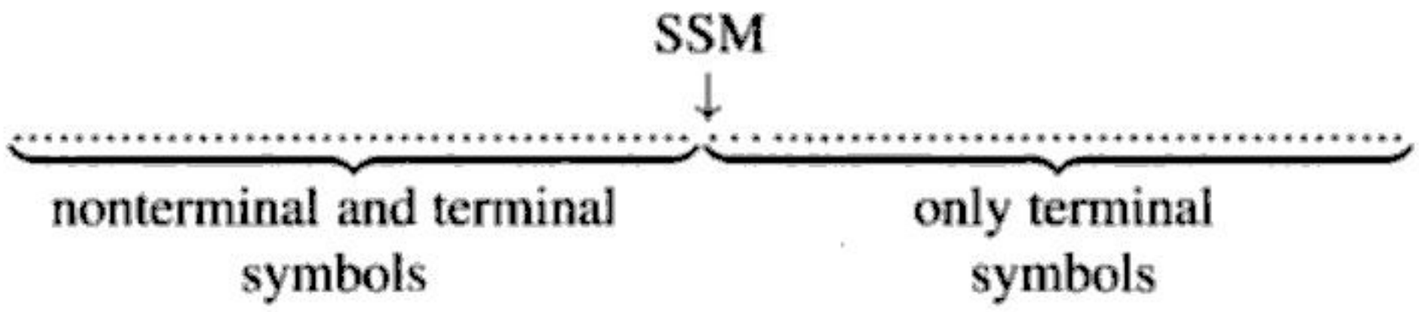
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 Example 6.16.

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

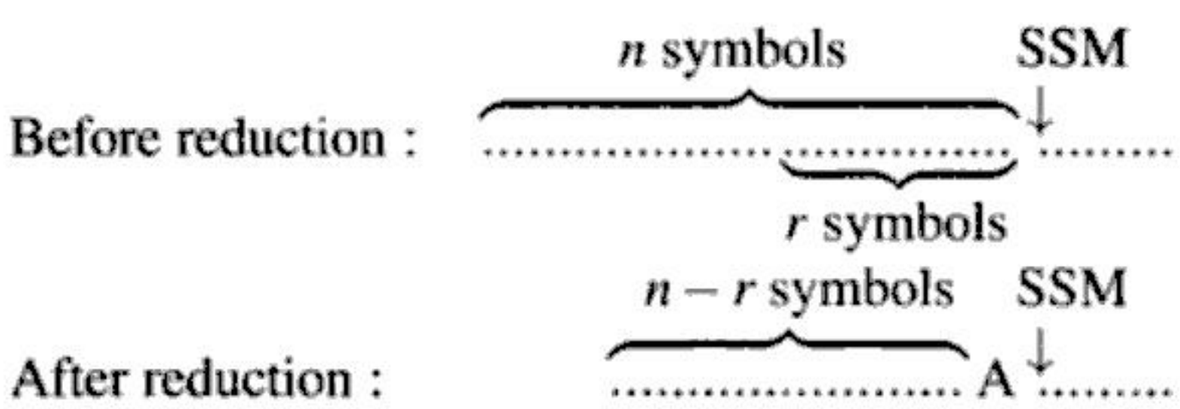
6.3.2 Bottom-Up Parsing

A bottom-up parser constructs a parse tree for a source string by applying a sequence of reductions to it. The source string is valid if it can be reduced to S, the distinguished symbol of G. If not, an error is detected and reported during the process of reduction. Bottom-up parsing proceeds in a left-to-right manner as follows: The parser applies reductions to the string located to the left of the *source string marker* (SSM). When it cannot apply any reductions, it advances the SSM by one character and tries to apply reductions to the new string that is located to the left of the SSM, and so on. Thus, a typical situation during bottom-up parsing can be depicted as follows:



To reach this situation, the parser would have performed some reductions and advanced the SSM over some symbols in the source string. Hence the string to the left of the SSM is composed of nonterminal and terminal symbols. The string to the right of the SSM is yet to be processed by the parser. Hence it consists of terminal symbols only.

We try out a naive approach to bottom-up parsing to understand the key issues involved in it. Let there be n symbols in the string to the left of the SSM. We try to reduce the last few symbols in this string to some nonterminal symbol, say A, by using one of the RHS alternatives in a production for A. Let this RHS alternative have r symbols in it. This reduction can be depicted as follows:



We advance the SSM if reduction is not possible for any nonterminal and any value of $r \leq n$. Since we do not know the value of r, we try the values $n, n-1, \ldots, 1$ for r. Algorithm 6.2 summarizes this approach.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Definition 6.7 (Handle) *A* handle *of a sentential form is the left-most simple phrase in it.*

We can identify a simple phrase in a sentential form of a simple precedence grammar by looking for a substring $s_1, s_2 \ldots s_r$, where each s_i is a terminal or nonterminal symbol, such that symbols in the string have the following precedence relations

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

Here s_1, s_r would be the first and last symbols to participate in the reduction. Needless to say that the string $s_1 s_2 \ldots s_r$ would match some RHS alternative in the grammar.

Example 6.19 (Simple phrase and handle) E + T is not a simple phrase of the string E + T * F according to Grammar (6.3) because the reduction E + T → E does not lead to the distinguished symbol of the Grammar (see Figure 6.10(a)). However, T * F is a simple phrase of the sentential form (see Figure 6.10(b)). It is also the handle of this sentential form.

Algorithm 6.2 is deficient because it does not identify simple phrases and does not perform reductions of handles. Algorithm 6.3 rectifies this deficiency.

Algorithm 6.3 (Bottom-up parsing)

1. Identify the handle of the current string form.
2. If a handle exists, reduce it according to the production of the grammar that has the handle as an RHS alternative. Go to Step 1.
3. If the current string form is 'S' then exit with success
   else report error and exit with failure.

However, most practical grammars are not simple precedence grammars, hence bottom-up parsing cannot be performed by using the notion of simple precedence. Example 6.20 illustrates this aspect.

Example 6.20 (Difficulties in using the notion of simple precedence) Grammar (6.3) is not a simple precedence grammar because the precedence relation between the symbols + and T is not unique: T * F is the handle in the sentential form E + T * F, hence T should be reduced prior to +. So + <· T. However, E + T is the handle in the sentential form E + T, so + ≐ T.

6.3.2.1 Operator Precedence Parsing

In Section 6.1.1, we defined an *operator grammar* as one whose productions do not have two or more consecutive nonterminal symbols in any RHS alternative. By induction, a sentential form cannot have two or more consecutive nonterminal symbols. This property simplifies the use of precedence in parsing—a parser can ignore the presence of nonterminal symbols and use precedences between operators for

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

operator. Hence the parser should perform a shift action if previous operator $\doteq$ current operator or previous operator $<\cdot$ current operator, whereas it should perform a reduce action if previous operator $\cdot>$ current operator. Because the only instance of the operator precedence relation $\doteq$ is '(' $\doteq$ ')', the handle is the substring '(..)' if the previous operator is ')'. In all other cases, the handle merely consists of the previous operator and its operands.

From these observations it is clear that apart from the current operator, only the previous operator needs to be considered at any point. Therefore, the parser employs a stack to store operators. It would push the current operator on the stack during a shift action and it would pop off the previous operator during a reduce action. To note the relation between operators and operands, each stack entry also accommodates information about the right operand of the operator stored in the entry. Information about the left operand of an operator, if any, 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.

Since operator precedence parsing ignores the nonterminal symbols in a string, it is not easy to build a parse tree for a source string. However, the parser can build an *abstract syntax tree*. The nodes in the tree would represent operators and operands of the expression and edges would represent relationships between them.

Algorithm 6.4 (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*.
complete : Boolean;

Functions

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

Input

An expression string enclosed between '⊢' and '⊣'.

1. TOS := SB − 1; complete := *false*;
   Push '⊢' on the stack;
   Set the SSM to point at the second source symbol.
2. If the current source symbol is an operand then
   { Build a node for the operand and store its pointer in TOS entry }
   x := *newnode* (*source symbol, null, null*);
   TOS.*operand_pointer* := *x*;
   Advance the SSM by one character;

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

produced by a scanner would consist of a set of tables of lexical units and a sequence of *tokens* for the lexical units occurring in a source statement. The scanner generated by LEX would be invoked by a parser whenever the parser needs the next token. Accordingly, each semantic action would perform some table building actions and return a single token.

Example 6.25 (Generation of a scanner using LEX) A specification input to LEX consists of four parts. Figure 6.15 shows three parts of a specification. The first part of the specification, which is enclosed by the strings %{ and %}, is used to define symbols that are 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 part is enclosed by the strings %% and %%. It contains the translation rules, i.e., string specifications and semantic actions. The third part contains auxiliary routines which can be used in the semantic actions specified in the translation rules.

```
%{
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 */}
```

Figure 6.15 A sample LEX specification

As discussed in Section 6.2, a token has the fields *lexical class* and *number in class*. LEX allows only the class code of a token to be returned by a semantic action. Hence each operator and keyword is made into a class by itself. The first three translation rules in the specification of Figure 6.15 specify the strings `begin`, `end`, and the assignment operator `:=` . Their semantic actions merely return the class codes for these strings. The fourth rule specifies an identifier. Its semantic actions would invoke the routine `enter_id`, which would enter the identifier in the symbol table, unless it already exists in the table, and return its entry number. The pair (`ID`, *entry #*) would form the token for the identifier string. To suit the LEX convention, *entry #* is put in the global variable `yylval`, and the class code `ID` is returned as the value of the call

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The notation called *regular expression* provides a concise method of specifying lexical units. A *deterministic finite state automaton* (DFA) is an abstract machine that is in one of its many states at any time and makes a transition to a specific new state when it is given a specific terminal symbol. If a lexical unit is specified as a regular expression, a DFA could be designed such that it would reach one of its *final states* on recognizing a lexical unit. Such a DFA can be used as a scanner. A DFA, and hence a scanner, can be automatically generated from a regular expression. LEX is a language processor development tool that generates scanners in this manner.

A production in the grammar can be used for two purposes—to *derive* a string from a nonterminal symbol, and to *reduce* a string to a nonterminal symbol. Accordingly, there are two fundamental approaches to the parsing of a string. *Top-down parsing* tries to generate a matching string from the distinguished symbol of the grammar, whereas *bottom-up parsing* focuses on finding whether the string can be reduced to the distinguished symbol. A *parse tree* of a string represents the sequence of derivations or reductions performed during its parsing; it represents the syntactic structure of the string. A grammar is ambiguous if more than one parse tree can be constructed for a string.

Top-down parsing is performed by making derivations according to productions of a grammar until a string that matches an input string can be generated. It is implemented through the steps of prediction making, matching and backtracking. Backtracking makes it inefficient and also compromises its diagnostic capability. A *recursive descent parser* is a top-down parser that avoids backtracking and its consequences. Bottom-up parsing is performed by applying reductions to an input string until it can be reduced to the distinguished symbol of the grammar. The notion of *precedence* can be used to decide when and how a reduction should be applied. An *operator grammar* and the notion of *operator precedence* can be used for efficient parsing of expressions. YACC is a language processor development tool that generates bottom-up parsers from a grammar.

TEST YOUR CONCEPTS

1. Classify each of the following statements as true or false:
   (a) If a program uses a variable named `alpha`, `alpha` is a terminal symbol.
   (b) Each valid sentence of a language can be derived from its distinguished symbol.
   (c) It is possible to find a sequence of reductions for a string from its abstract syntax tree.
   (d) A DFA can have more than one transition out of a state for a source symbol.
   (e) The notion of precedence can be used to avoid ambiguity in a grammar.
   (f) A regular expression can be used to specify nested structures.
   (g) Grammars containing left recursion are amenable to top-down parsing.
   (h) Backtracking may have to be performed during bottom-up parsing.
   (i) An operator precedence grammar has fewer precedence relations than an equivalent simple precedence grammar.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(ii) $< id > * (< id > + < id >) / < id >$

10. Compare and contrast recursive descent parsing with LL(1) parsing. What grammar forms are acceptable to each of these?
11. Construct an operator precedence matrix for a grammar for expressions containing arithmetic, relational and boolean operators.
12. Given the following operator grammar

```
S   ::=   ⊢A⊣
A   ::=   VaB | ε
B   ::=   VaC
C   ::=   VbA
V   ::=   <id>
```

    (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.

13. The keywords **if**, **then** and **else** are operators in the grammar of the **if** statement given in Problem 5(c). 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 (2002) and Aho, Lam, Sethi and Ullman (2007) discuss parsing techniques in detail.

1. Aho, A. V., M. Lam, R. Sethi, and J. D. Ullman (2007): *Compilers – Principles, Techniques and Tools*, second edition, Addison-Wesley, Reading.
2. Barrett, W. A. and J. D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.
3. Dhamdhere, D. M. (2002): *Compiler Construction – Principles & Practice*, second edition, 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 7

Compilers

In Chapter 2, we have seen that a compiler bridges the *semantic gap* between a programming language domain and an execution domain and generates a *target program*. The target program may be a machine language program or an *object module* discussed earlier in Chapter 5.

The compiler performs two kinds of tasks while bridging the semantic gap: providing diagnostics for violations of programming language semantics in a source program, and generating code to implement meaning of a source program in the execution domain. Diagnostics are provided during scanning, parsing, and semantic analysis of a program. In Chapter 6, we have discussed scanning and parsing, and the LEX and YACC tools that can be used for generating scanners and parsers. The attributes of nonterminal symbols provided in YACC can be used for performing semantic actions that provide diagnostics and generate code.

We discuss details of code generation in this chapter. To understand the key issues that need to be addressed during code generation, we first discuss features of programming languages that contribute to the semantic gap between a programming language domain and an execution domain. These features are:

- Data types
- Data structures
- Scope rules
- Control structure

We then discuss the techniques that are used to bridge the semantic gap effectively. These techniques are grouped into techniques for memory allocation to data structures, for handling scope rules, for generating code for expressions and control structures, and for optimizing the code generated for a program.

7.1 CAUSES OF A LARGE SEMANTIC GAP

As discussed in Chapter 2, the *semantic gap* refers to the difference between semantics of two domains. A compiler is concerned with the programming language domain of its source language and the execution domain of its *target machine*, which is either a host computer or a *virtual machine* running under some operating system. Hence the semantic gap is caused by those features of the programming language that do not have corresponding features in the target machine. In this section, we discuss four such features and mention how a compiler may bridge the semantic gap caused by them. Note that the output of a compiler is a *target program*, which may be a machine language program for the target machine mentioned above, or an *object module* discussed earlier in Chapter 5.

Data types

Definition 7.1 (Data type) *A* data type *is the specification of* (*i*) *values that entities of the type may have, and* (*ii*) *operations that may be performed on entities of the type.*

We refer to these values and operations as *legal values* and *legal operations* of a type. Legal operations of a type typically include an assignment operation and a set of data manipulation operations. A programming language provides a few *standard* data types whose definitions are a part of the language specification, e.g., most languages support the data type *boolean*, which has the legal values *true* and *false* and the legal operations *and*, *or*, *not* and assignment. A programming language may also allow a user to define her own data types. Such data types are called *user-defined* data types. The compiler must check whether variables of a type are assigned legal values and whether variables and values of the type are manipulated through legal operations, and issue diagnostic messages when these requirements are not met.

A programming language may specify rules for *type conversion*, whereby a legal value of one data type can be converted into a legal value of another data type. It would typically permit conversion between numerical types such as *integer* and *real*, but may forbid conversion between other types. When an assignment operation or an expression in a program contains values of different numerical types, the compiler should apply the type conversion rules to decide which of the values should be converted to equivalent values in other types. Once these decisions are made, it should use an appropriate instruction of the target machine to implement each of the operations. When an execution domain does not contain values or operations that correspond to those of a data type, the compiler must decide how to represent the value of a type and what instructions or sequences of instructions to use for performing the operations of a type.

Examples 7.1 and 7.2 illustrate how a compiler handles data types. In Example 7.1, the execution environment has features that directly support the values and operations of a type, whereas Example 7.2 illustrates handling of a type where such features are absent.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

elements of `info`. Thus the PL/1 program of Example 7.7 may execute slower than the Pascal program of Example 7.6 (see Section 7.5.3.4 for more details). However, the PL/1 program provides greater flexibility because the procedure can handle any two-dimensional array irrespective of its dimension bounds. In fact, in different calls it can handle arrays having different dimension bounds. Hence we can draw the following inference: 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 7.3 (Static binding) *A* static binding *is a binding performed before the execution of a program begins.*

Definition 7.4 (Dynamic binding) *A* dynamic binding *is a binding performed after the execution of a program has begun.*

Use of static binding leads to more efficient execution of a program than use of dynamic binding. We shall discuss static and dynamic binding of memory in Section 7.5.1 and dynamic binding of variables to types in Chapter 8.

7.3 DATA STRUCTURES USED IN COMPILERS

Two kinds of data structures are used during compilation and execution of a program. A *search data structure* is used to maintain information concerning attributes of entities in the program. Search efficiency is the key criterion in its design. An *allocation data structure* is used to decide which area of memory should be allocated to an entity. Speed of allocation or deallocation and efficiency of memory utilization are the important criteria in the design of allocation data structures. In Section 2.4 we discussed how symbol tables are designed to provide search efficiency. In this section we discuss two allocation data structures called *stack* and *heap*.

A compiler uses both search and allocation data structures while compiling a source program. It uses a search data structure to constitute a table of information, such as the symbol table and the constants' table. If the program being compiled contains nested blocks or nested procedures and functions, it would use an allocation data structure while allocating these tables. During execution, the target code of the program may use either search or allocation data structures in accordance with its logic. The target code would also invoke routines in the *run time environment* of the programming language. As we shall see in Section 7.5, some of these routines may use allocation data structures while allocating memory to the program's data.

7.3.1 Stack

The last-in-first-out nature of the stack is useful for handling nested use of entities. The stack data structure and the extended stack model was described earlier in Section 4.6.6, and its use for expansion of nested macro calls was discussed. As we shall

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Scope of a variable

Entities declared in a block must have unique names. These entities can be accessed only within that block. However, a program may contain a nesting of blocks and entity names may not be unique in the program. Hence the specification of a language contains rules for determining the *scope* of a variable, which consists of those parts of a program wherein the variable can be accessed.

Let a data declaration in block b that uses the name $name_i$ create a variable var_i. The scope rules of a block-structured language specify that

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. We use the following notation 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.

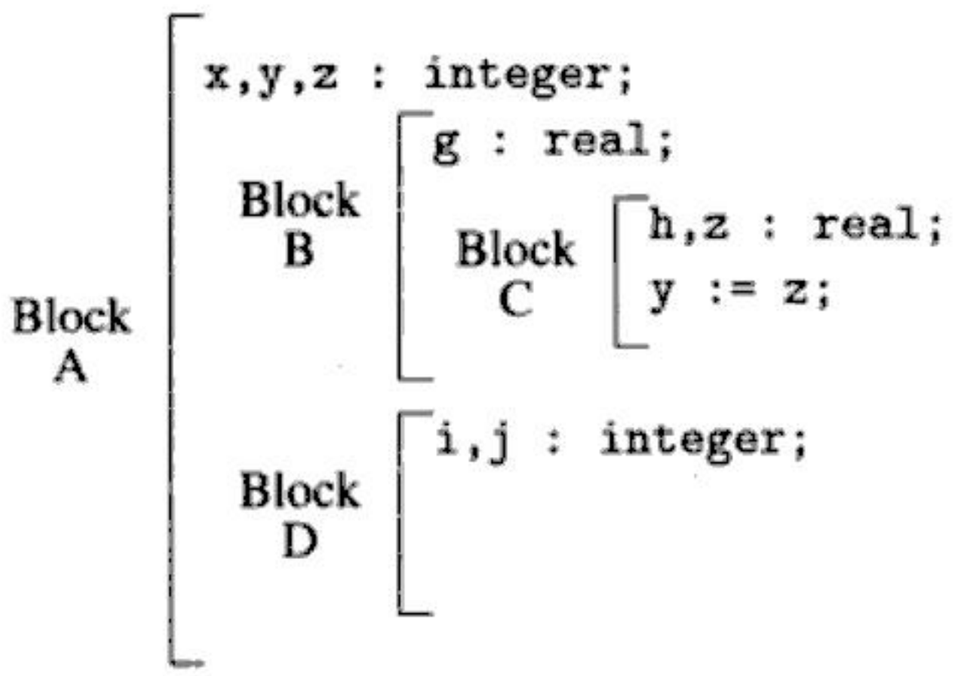


Figure 7.4 A block structured program

Example 7.11 (Local and nonlocal variables of blocks) Consider the block-structured program in Figure 7.4. The variables that are accessible within the various blocks are as follows:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

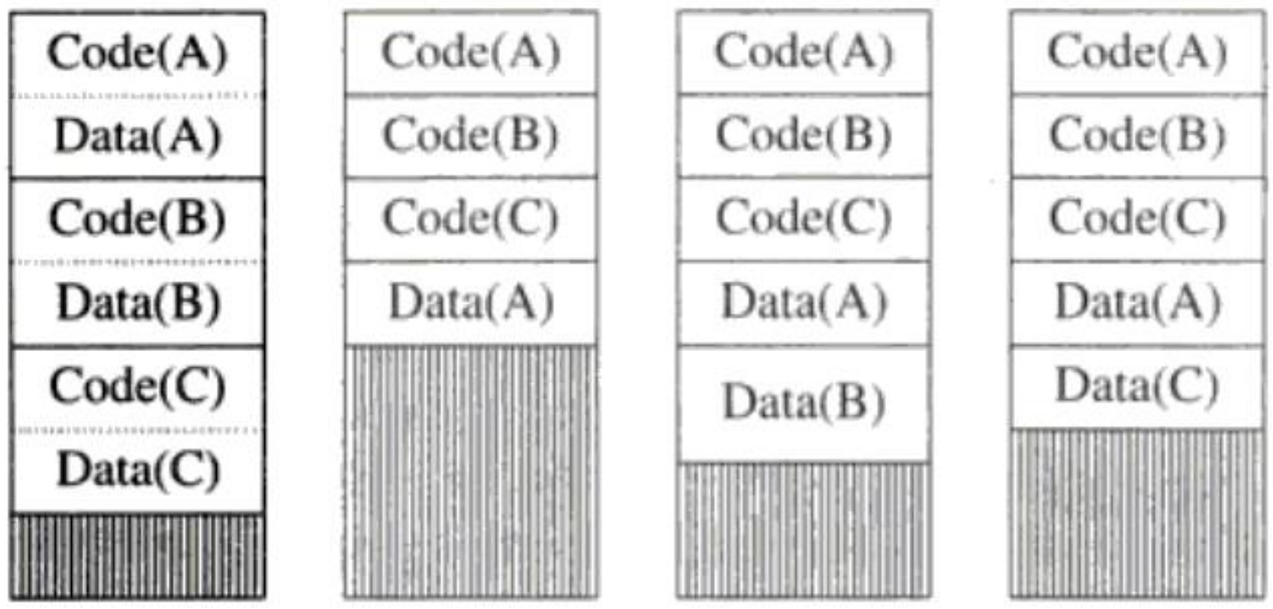


Figure 7.6 (a) Static memory allocation, (b)–(d) Dynamic memory allocation

scheme illustrated in Example 7.13. Because entry and exit from procedures follow the last-in-first-out rule, automatic dynamic allocation is implemented by using a *stack*. When a procedure is entered during the execution of a program, a record is created in the stack to contain its variables and a pointer is set to point at this record. Individual variables of the procedure are accessed by using displacements from this pointer. The record is popped off the stack when the procedure is exited. Details of this scheme are discussed later in Section 7.5.3. In *program controlled dynamic allocation*, a program can allocate or deallocate memory at any time during its execution. Program controlled dynamic allocation is implemented by using the *heap* data structure, which was discussed in Section 7.3.2.

Dynamic allocation provides two significant advantages. Recursion is easy to implement because a recursive call on a function or subroutine leads to allocation of a separate memory area for its new activation. We discuss recursion later in Section 7.5.3.3. Data structures whose sizes become known only dynamically can also be handled naturally. For example, if an array is declared as `a [m,n]`, where `m` and `n` are variables, size of the memory area to be allocated can be determined from the values of `m` and `n`. However, in both automatic dynamic allocation and program controlled dynamic allocation, address of the memory area allocated to a variable cannot be known at compilation time. Hence a variable is accessed through a pointer. This arrangement may make a program using dynamic memory allocation execute slower than a program that uses static memory allocation.

7.5.2 Dynamic Memory Allocation and Access

This section discusses only automatic dynamic memory allocation. It is implemented by adapting the extended stack model discussed earlier in Section 4.6.6.1. Each record in the stack is used to accommodate variables of one activation of a block, hence we call it an *activation record* (AR). An activation record has the form shown in Figure 7.7. It contains two reserved pointers. As discussed later in this section,

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

of a block structured language, when *b_use* is in execution, *b_defn* must be active. Hence AR_{b_defn} would exist in the stack, and the nonlocal variable *nl_var* should 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}. The compiler needs to set up an arrangement by which the start address of AR_{b_defn} can be determined at any time during execution of the generated code. We discuss this arrangement in the following.

We use the following terminology for a block and its enclosing blocks: A *textual ancestor* or *static ancestor* of block *b_use* is a block that encloses block *b_use*. The block immediately enclosing *b_use* is called its level 1 ancestor. A level *m* ancestor is a block that immediately encloses the level $(m-1)$ ancestor. The *level difference* between *b_use* and its level *m* ancestor is *m*. $s\_nest_{b\_use}$ represents the static nesting level of *b_use*, that is, nesting level of block *b_use* in the program.

Access to nonlocal variables is implemented by using the second reserved pointer in the activation record, which has the address 1(ARB). This pointer is called the *static pointer* (see Figure 7.7). Step 5 of the actions performed at block entry (see Table 7.1) sets the static pointer in the activation record of a block to point at the activation record of the block's static ancestor. If a statement in the block accesses a nonlocal variable *nl_var* declared in a level *m* ancestor of the block, the compiler would know the value of *m* from the searches made in the symbol table during name resolution for *nl_var* (see Example 7.12). Hence it would generate the following code to access *nl_var*:

1. r := ARB; where r is some CPU register.
2. Repeat Step 3 *m* times.
3. $r := 1(r)$; i.e., load the static pointer of the activation record to which register r is pointing into CPU register r.
4. Access *nl_var* by using the address $< r > + d_{nl\_var}$.

(7.1)

Thus the code traces the list formed by static pointers of activation records until the activation record of the level *m* ancestor is reached. It involves *m* indirections through static pointers. Example 7.15 illustrates access to nonlocal variables in the the program of Example 7.14.

Example 7.15 (Access to nonlocal variables) Figure 7.9 shows memory allocation for the program of Example 7.14. When block B was entered, the static pointer in AR_B would have been set to point at the start of AR_A. When block C is entered, Step 5 of the actions performed at block entry (see Table 7.1) sets the static pointer in AR_C to point at the start of AR_B. The code generated to access variable x in the statement `x := z;` is now

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
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]
```

Figure 7.15 Target code for an array reference

the dope vector. The generated code would use d_{DV} to access the contents of the dope vector for checking the validity of subscripts and computing the address of an array element. Example 7.20 illustrates how the dope vector for an array with dynamic bounds would be used during execution.

Example 7.20 (Accessing an element of an array with dynamic bounds) Figure 7.16 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 of alpha are not known during compilation, the dope vector must exist during execution. Hence it is allocated in the activation record of the block. d_{alpha_DV}, its displacement in the activation record, is known at compilation time. Array alpha is allocated when the block is entered and the address of the base element of the array is recorded in its dope vector. The code generated for access of the array element alpha[i,j] uses d_{alpha_DV} to access the contents of the dope vector. For example, the word with the address 4 (ARB) is the *address* field in the dope vector which would contain address of the base element of alpha during execution and 6 (ARB) would contain information about its first dimension.

Programming languages like C, C++ and Java do not allocate and access multi-dimension arrays the way we have discussed so far. Instead, they use an arrangement having many one-dimensional arrays. For example, a column-major arrangement for a 5×10 array would have a one-dimensional array containing 10 pointers, one pointer for each column, where each pointer points to a one-dimensional array having

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A multi-pass organization requires the use of an *intermediate representation* (IR) of a program, which consists of an *intermediate code* and a set of tables. This section describes three widely-used intermediate codes for expressions.

7.6.3 Postfix Notation

Expressions in most programming languages are written in the *infix notation* wherein a binary operator appears between its two operands, e.g., `a*b`. By contrast, in the *postfix* notation each operator appears immediately after its last operand, e.g., `ab*`. Thus, a binary operator op_i appears after its second operand. If any of its operands is itself an expression involving some operator op_j, op_j would have to appear before op_i in the string. For simplicity, we call a string written in the postfix notation a *postfix string*. A postfix string has the useful property that operators in it can be evaluated in the order in which they appear in the string. Example 7.25 illustrates this property.

Example 7.25 (Postfix notation) Consider an arithmetic expression using the +, * and ↑ (exponentiation) operators in the infix notation and the corresponding postfix string

```
                  2   1   5   4   3
Source string : |- a + b * c + d * e ↑ f -|                    (7.6)

                  1 2         3 4 5
Postfix string : |- a b c * + d e f ↑ * + -|
```

The boxed numbers appearing above the operators indicate their bottom up evaluation order. The second operand of the operator '+' marked [2] in the source string is the expression `b*c`. Hence in the postfix string its operands `b`, `c` and its operator '*' appear immediately before the '+' marked [2]. Note that the order of appearance of the operators in the postfix string matches their bottom-up evaluation order.

The postfix string is a popular intermediate code in non-optimizing compilers because of the ease with which it can be both obtained from the source string and used for generating target code. Because operators in a postfix string appear in their bottom-up evaluation order and because an operator appears *after* its operands, code can be generated for it immediately. Its operands are the last few operands among the operands in the string. If one of its operands is a partial result produced by another operator, the operator and its operands would appear to its left in the string so code would have been generated for that operator already. Thus operands appearing in the postfix string are used in the last-in-first-out order so we can use a stack to store them until they are used for code generation.

The complete scheme for code generation from a postfix string is as follows: The postfix string is processed in a left-to-right manner. If the next symbol in it is an operand, an *operand descriptor* is built and it is pushed on a stack of operand descriptors. If the next symbol is an operator with arity k, k descriptors are popped off the stack and one or more instructions are generated for evaluating the operator.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

appear in a desirable 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;
```

Figure 7.29 Procedure *evaluation_order*

If the RR label for the root node is a number q, use of the evaluation order determined by Algorithm 7.1 would result in code that would not save any partial result(s) to memory if at least q CPU registers are available. It thus provides the most efficient way to evaluate the expression. If the number of available registers is $< q$, some partial results would have to be saved in memory. However, no other evaluation order will require fewer partial results to be saved in memory so use of this evaluation order still leads to an efficient code. A proof of this property is beyond the scope of our discussion.

Example 7.29 (Working of Algorithm 7.1) Figure 7.30 shows the expression tree for the string `f+(x+y)*((a+b)/(c-d))`. The boxed number adjoining an operator node indicates that operator's position in the source string. We will use this number to refer to the operator. Algorithm 7.1 first assigns RR labels to the nodes in the tree. The RR label of each node is shown as the superscript of the operand or operator at that node. Some significant steps in computation of RR labels are as follows: The node of `a` has the label 1 because it is a leaf node that is the left child of another node, so its value would have to be loaded in a register before the operation represented by its parent node is performed. The node of `b` has the label 0 because it is the right child of a node, so its value does not have to be loaded in a register. Hence the node of operator [4] has the label 1. Similarly, nodes of operators [2] and [6] have the labels 1 and 1, respectively. The node of operator [5] has the label 2 because its child nodes have labels 1 and 1. Nodes of operators [3] and [1] have labels 2 and 2, respectively, because the child nodes of each of them have labels 1 and 2. The algorithm determines the evaluation order of the operators as follows: It calls procedure *evaluation_order* with the root node, i.e., operator [1], as the parameter. *evaluation_order* finds that RR(node for `f`) < RR(node for operator [3]), where `f` is its left child, so it decides that the right child of the node should be evaluated before its left child. It now makes a recursive call with node [3] as the parameter. Here it finds RR(node for operator [5]) > RR(node for operator [2]), so it makes a recursive call with node [5] as the parameter. At node [5] also it decides to visit the right child before the left child. These decisions lead to the postfix string `cd-ab+/xy+*f+`. Thus, the evaluation order is [6], [4], [5], [2], [3], [1].

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$Eval_i$: 'true' only if b_i contains an evaluation of e that is not followed by assignments to any operands of e

$Modify_i$: 'true' only if b_i contains assignment(s) to some operand(s) of e.

$Eval_i$ and $Modify_i$ depend entirely on the computations situated in b_i, so they are called local properties of block b_i. $Avail\_in_i$ and $Avail\_out_i$ are 'global' properties because they depend on properties at predecessors of b_i. Following (7.8), the global properties $Avail\_in_i$ and $Avail\_out_i$ are computed by using the following equations

$$Avail\_in_i = \Pi_{all\ b_j\ in\ pred\ (b_i)}\ Avail\_out_j \quad (7.9)$$

$$Avail\_out_i = Eval_i + Avail\_in_i\ .\ \neg Modify_i \quad (7.10)$$

where $\neg$ is the boolean negation operation and $\Pi_{all\ b_j\ in\ pred\ (b_i)}$ is the boolean 'and' operation over all predecessors of b_i. Equation (7.9) implies condition 2 of (7.8). It ensures that $Avail\_in_i$ would be true only if *Avail_out* is true for all predecessors of b_i. Equation (7.10) implies condition 1 of (7.8). Equations (7.9) and (7.10) are called *data flow equations*.

Every basic block in G_P has a pair of equations analogous to (7.9)–(7.10). 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 of data flow analysis which assigns some initial values to *Avail_in* and *Avail_out* of all basic blocks in G_P and iteratively recomputes them according to Equations (7.9)–(7.10) until they converge onto consistent values.

The initial values are

$$Avail\_in_i = \begin{cases} false & \text{if } b_i \text{ is the entry node } n_0 \\ true & \text{otherwise} \end{cases}$$

$$Avail\_out_i = true$$

After solving the system of equations (7.9)–(7.10) for all blocks, global common subexpression elimination is performed as follows: An evaluation of expression e is 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 7.40 illustrates application of this method.

Example 7.40 (Global common subexpression elimination) Available expression analysis for the control flow graph of Figure 7.36 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

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

12. Watson, D. (1989): *High Level Languages and their Compilers*, Addison-Wesley, Reading.

CHAPTER 8

Interpreters

In Chapter 2 we defined an interpreter as a language processor that bridges the execution gap of a program written in a programming language without generating a *target program*. This characteristic makes interpreters attractive for use in a program development environment because it avoids the overhead of compilation between successive trial runs of a program. An interpreter can itself be coded in a programming language, which makes it portable.

An interpreter performs interpretation of a source program by using the *interpretation cycle* repeatedly. In each execution of this cycle, it analyzes the next statement of the source program to find its meaning and then performs actions that implement the meaning. This arrangement incurs substantial overhead during interpretation, which makes interpretation slow. An *impure interpreter* is a hybrid scheme used to speed up interpretation of programs. In this scheme, some elements of compilation are integrated with the interpreter—the source program is processed to build an intermediate code which can be interpreted faster than the source language form of the program. In this chapter we discuss how interpreters perform actions indicated in a source program without generating a target program and how impure interpreters achieve faster interpretation.

The Java language environment incorporates many of the considerations discussed in Chapter 1. It uses a *virtual machine* to provide portability of programs. It provides both an interpreter and a just-in-time compiler to make its working adaptive and provides the capability to download and execute programs over the Internet. We discuss the Java virtual machine, the *Java bytecode* which is the machine language of the virtual machine, and working of the Java *bytecode verifier* that provides security even when unknown Java programs are downloaded and executed.

8.1 BENEFITS OF INTERPRETATION

Benefits of interpreters arise from the fact that the interpreter does not generate a target program. It is simpler to develop an interpreter for a programming language than to develop a compiler for it because interpretation does not involve code generation (see Section 8.2). This simplicity makes interpretation more attractive when programs or commands are not executed repeatedly. Hence interpretation is a popular choice for commands to an operating system or an editor. The user interfaces of many software packages prefer interpretation for similar reasons.

Use of interpretation avoids the overhead of compilation of a program. However, during interpretation every statement is subjected to the interpretation cycle, which is expensive in terms of CPU time. Thus, a trade-off exists between compilation of a program and its interpretation. We use the following notation to illustrate this trade-off:

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 analyze 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 and t_e to be almost equal, i.e., $t_c \cong t_i$. t_e is actually the execution time of the target code for a statement generated by the compiler; it can be several times smaller than t_c. Let us assume $t_e = \frac{t_c}{20}$.

To help us decide the conditions under which it is beneficial to use interpretation, Example 8.1 performs a quantitative comparison of compilation and interpretation in a program development environment.

Example 8.1 (Compilation and interpretation in a program development environment) Let $size_P$ and $stmts\_executed_P$ represent the number of statements in a program P and the number of statements executed in some execution of P, respectively. Let $size_P$ = 200. For a specific set of data, let program P execute as follows: 20 statements are executed for initialization purposes. It is followed by 10 iterations of a loop containing 8 statements, followed by the execution of 20 statements for printing the results. Thus, $stmts\_executed_P = 20 + 10 . 8 + 20 = 120$. Thus,

Total execution time using compilation
$= 200 . t_c + 120 . t_e$
$\cong 206 . t_c$.
Total execution time using interpretation
$= 120 . t_i$
$\cong 120 . t_c$.

Following Example 8.1, it is better to use an interpreter if a program is modified between executions, and $stmts\_executed_P < size_P$. These conditions are satisfied during program development, hence interpretation should be preferred.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- The program violates access restrictions, e.g., by accessing *private* data
- The program has type-mismatches whereby it may access data in an invalid manner
- The program may have stack overflows or underflows during execution.

It accepts the class file only if the program does not have any of these flaws; otherwise, it aborts interpretation of the program that wished to include this class file.

Use of the impure interpretation scheme provides efficient interpretation because the Java virtual machine does not perform lexical, syntax or semantic analysis during interpretation of a program. However, interpretation is slower than compilation. Hence the Java language environment provides the two compilation schemes shown in the lower half of Figure 8.4. The Java *Just-In-Time* compiler compiles parts of the Java bytecode that are consuming a significant fraction of the execution time into the machine language of the computer to improve their execution efficiency. It is implemented using the scheme of *dynamic compilation* discussed in Section 1.4.7; it has the benefit that the overhead of compilation is not incurred for those parts of the program that do not consume much of the execution time. After the just-in-time compiler has compiled some part of the program, some parts of the Java source program has been converted into the machine language while the remainder of the program still exists in the bytecode form. Hence the Java virtual machine uses a *mixed-mode execution* approach—it interprets those parts of the program that are in the bytecode form, and directly executes the machine language of those parts that were handled by the just-in-time compiler.

The other compilation option uses the Java native code compiler shown in the lower part of Figure 8.4. It simply compiles the complete Java program into the machine language of a computer. This scheme provides fast execution of the Java program; however, it cannot provide any of the benefits of interpretation or just-in-time compilation described earlier.

8.3.1 Java Virtual Machine

A Java compiler produces a binary file called a *class file* which contains the bytecode for a Java program. The Java virtual machine loads one or more class files and executes programs contained in them. To achieve it, the JVM requires the support of the class loader, which locates a required class file, and a bytecode verifier, which ensures that execution of the bytecode would not cause any breaches of security. Hence these modules are considered to be components of the JVM. The JVM also uses a library of methods in the native code of the computer to assist it in the interpretation of bytecode instructions. However, the native methods may vary across implementations of the JVM.

The Java virtual machine is a *stack machine*. The hypothetical computer we used in Chapter 3 performed computations in general-purpose registers. By contrast, a stack machine performs computations by using the values existing in the top few

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

1. `debug unit` (<*file_number*>, <*rmlist_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*> <*list_of_actions*>

   Indicated actions are executed when statement bearing <*label*> is encountered during execution. The possible actions are:

`trace on`	:	Trace is enabled
`trace off`	:	Trace is disabled
`display` <*list*>	:	Values of variables in the list are written in the debug file

Figure 9.4 Trace and dump facilities in Fortran

2. Examining or changing values at a breakpoint
3. Stepping through a program's execution in a statement-by-statement manner.

A *breakpoint* is a place in a program where its execution should be suspended to enable a debug conversation. It can be specified statically by identifying a source statement as in the `at` command of Figure 9.4, in which case the debug conversation is initiated when the program's execution reaches the statement. Alternatively, it can be specified through a condition, in which case the debug conversation is initiated whenever the condition is satisfied during the program's execution. Use of dynamic specification incurs more overhead during the program's execution because the condition would have to be evaluated repeatedly. However, it is the most effective method of capturing certain kinds of errors. For example, if the user finds that a variable `xyz` has somehow been assigned the "wrong" value 100, it can ask a debug conversation to be initiated when the condition `xyz = 100` is satisfied. If she wishes to examine variables' values in the 50^{th} iteration of a loop, she can specify a condition on the control variable of the loop.

At a breakpoint, the user can examine values of interest, and even change values of variables to facilitate further debugging or to check out a proposed bug fix. To study how the program could have gone wrong, she can set a breakpoint that would occur before the program has assigned the wrong value and then execute the program in a statement-by-statement manner to observe its behaviour.

Example 9.6 (Dynamic debugging) Figure 9.5 illustrates use of dynamic debugging facilities supported by many Basic interpreters. The programmer can set breakpoints at

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the second developer's copy, it would contain changes in function `delete_info` but none in function `add_info`. Thus changes made by the first developer would be lost.

The consistency issue caused by simultaneous modifications to a file is resolved by merging a modified file with the file stored in the file system, rather than by overwriting the file. In Example 9.10 changes made by the first developer would be merged with the original file and changes made by the second developer would be merged with the file resulting from the first merge operation. This way, changes made by the first developer would survive. Inconsistencies can still arise if the developers had made changes in the same statements. The system can warn the second developer of this danger when she executes her commit operation.

The Source code control system (SCCS) was the first version control system that saw extended use under Unix. Concurrent versions system (CVS) allowed many developers to edit the same file simultaneously; however, it did not use atomic updates. The version control system Subversion (SVN) is widely used in the open source community. It uses atomic updates.

9.2.6 Program Documentation Aids

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, cross reference information, etc.

9.2.7 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 new 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 a debug monitor an inserted statement checks whether a breakpoint condition is satisfied and opens a debug conversation. Example 9.11 illustrates how program instrumentation is performed.

Example 9.11 (Program instrumentation) 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 instru-

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

9.4.1 Testing Assertions

A debug assertion is a relation between the values of program variables. It can be associated with a program statement. The debug monitor verifies the assertion when execution reaches that statement. Execution of the program continues if the assertion is fulfilled; otherwise, a debug conversation is opened so that the user can perform actions for locating the cause of the program malfunction. Use of debug assertions eliminates the need to produce voluminous information for debugging purposes.

9.5 PROGRAMMING ENVIRONMENTS

A *programming environment* is a software system that provides integrated facilities for program creation, editing, execution, testing and debugging. Its use avoids use of distinct software tools during different steps in program development, thereby providing convenience and enhancing programmer productivity. A software environment 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*, typically an *abstract syntax tree*. The compiler (or interpreter) and the debug monitor share the intermediate representation. If a compiler is used, it is activated after the editor has converted a statement to intermediate representation. The compiler works incrementally to generate code for the statement. This way, execution or interpretation of the program can start immediately after the last statement has been input. The programmer can interrupt execution of the program at any time to either enter the debug mode or return to the editor, 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 arrangement 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.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 which application function should 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.

9.7 SUMMARY

Users and computational activities need support of various housekeeping functions for maintenance and interfacing of programs. These functions are performed by system programs called *software tools*. A software tool interfaces a program with the entity from which it draws its input and with the entity that uses its output. In this chapter we discussed various software tools used in program development, and user interfaces.

A user can employ a variety of software tools in program development. These

include familiar tools such as editors and some specialized software tools. A *programming environment* provides integrated facilities for program creation, editing, execution, testing and debugging. Its use avoids having to use distinct software tools during different steps in program development, thereby providing convenience and enhancing programmer productivity. A *test data generator* analyzes a program and generates test data that would exercise various execution paths in it, thereby eliminating the drudgery of testing and making testing more comprehensive. A *debug monitor* helps in localization of errors by allowing a programmer to halt execution of a program at interesting points and examine values of variables to detect bugs and diagnose their causes. Its use improves the productivity in program debugging. A *profile monitor* helps in performance tuning of a program by collecting information concerning the program's behaviour during execution. A *source code management system* helps to ensure consistency of a program during debugging and modification. A *version control system* helps in maintenance of a program that has several versions. We discussed the design of editors, debug monitors, and programming environments.

A *user interface* simplifies the interaction of a user with an application by providing means for conducting *command dialogs*. We discussed principles of command dialog design and structure of user interfaces.

TEST YOUR CONCEPTS

1. Classify each of the following statements as true or false:
   (a) An infeasible path is simply a path that may not be traversed during an execution of a program.
   (b) To check whether values are used correctly in a loop, one should set a breakpoint at a statement that is executed after exiting the loop.
   (c) A source code management system is used to decide whether a modification is correct.
   (d) A debug monitor uses program instrumentation.
   (e) To handle queries of an ad hoc kind, one should develop a program generator rather than an interpreter.

EXERCISE 9

1. Comment on the statement "Dynamic debugging is easier to implement in interpreters than in compilers".
2. Comment 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.

PART II

OPERATING SYSTEMS

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CHAPTER 13

File Systems

Computer users store programs and data in files so that they can be used conveniently and preserved across computing sessions. A user expects the file system to provide many facilities—convenient and fast access to files, reliable storage of files, and means to share files with co-workers. The operating system must fulfill these expectations and also ensure efficient use of I/O devices.

Operating systems organize file management into two components. A *file system* provides facilities for creating and efficiently manipulating files, for ensuring reliability of files when faults such as power outages or I/O device malfunctions occur, and for specifying how files are to be shared among users. The *input output control system* (IOCS) provides efficient access to data stored on I/O devices, and efficient performance of I/O devices.

The file system provides different *file types*, each with its own characteristics, and a *directory structure* for convenient grouping of related files. It allocates disk space when a file is created or extended. It ensures reliability through two means. *Recovery techniques* are used to restore files to a consistent state when failures occur. *Fault tolerance techniques* are used to ensure that the file system can continue its operation even when failures occur. The IOCS implements I/O operations using the DMA, and uses *disk scheduling* to improve the rate at which a disk can perform I/O operations. It uses the techniques of *buffering*, *blocking* and *caching* of data to speed up I/O operations in a process. In this chapter we discuss file systems and IOCS in that order.

13.1 OVERVIEW OF FILE PROCESSING

We use the term *file processing* to describe the general sequence of operations of opening a file, reading data from the file or writing data into it, and closing the file. Management of file processing activities is structured in two components:

- File system
- Input output control system (IOCS).

A file system provides several *file types* (see Section 13.2) and operations that can be performed on files of each file type. Each file type provides its own abstract view of data in a file—we call it a *logical view* of data. The IOCS organizes a file's data on an I/O device in accordance with its file type. It is the *physical view* of the file's data. The mapping between the logical view of the file's data and its physical view is performed by the IOCS. The IOCS also provides an arrangement which speeds up a file processing activity—it holds some data from a file in memory areas organized as *buffers*, a *file cache* or a *disk cache*. When a process performs a `read` operation to get some data from a file, the IOCS takes the data from a buffer or a cache if it is present there. This way, the process does not have to wait until the data is read off the I/O device that holds the file. Analogously, when a process performs a `write` operation on a file, the IOCS copies the data to be written in a buffer or in a cache. The actual I/O operations to read data from an I/O device into a buffer or a cache, or to write it from there into an I/O device, are performed by the IOCS in the background.

13.1.1 File System and the IOCS

A file system views a file as a collection of data that is *owned* by a user, can be *shared* by a set of authorized users, and has to be *reliably stored* over an extended period of time. A file system gives users freedom in naming their files so that a user can give a desired name to a file without worrying whether it conflicts with names of other users' files; and it provides privacy by protecting against interference by other users. The IOCS, on the other hand, views a file as a repository of data that need to be *accessed speedily* and are stored on an I/O device that needs to be *used efficiently*.

The file system and the IOCS form a hierarchy. The file system provides an interface through which a process can perform open, read/write and close operations on files. Its modules handle protection and sharing of files and pass on read/write requests for file data to the IOCS. The IOCS modules ensure efficient operation of I/O devices and efficient file processing in each process.

Data and metadata A file system houses two kinds of data—data contained within files, and data used to access files. We call the data within files *file data*, or simply *data*. The data used to access files is called *control data*, or *metadata*, e.g., data in the directory entry of a file. As discussed later in this chapter, metadata also play a role in implementing file operations.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

SYSTEMS PROGRAMMING

This book is designed for undergraduate and postgraduate courses on Systems Programming. It offers in-depth treatment for the fundamental concepts in systems programming and different kinds of system software. Furthermore, it stresses on the use of system programming concepts in designing various system software.

Key Features

- Dedicated chapters on Interpreters, Scanners and Parsers
- Detailed coverage of topics such as Assemblers, Macro Processors, Java Language Environment and Operating Systems
- Includes an overview of contemporary trends in system software
- Numerous solved examples and exercises interspersed in the text

The McGraw-Hill Companies

Mc Graw Hill Higher Education

Visit us at : www.tatamcgrawhill.com

ISBN-13: 978-0-07-133311-5
ISBN-10: 0-07-133311-8

9 780071 333115