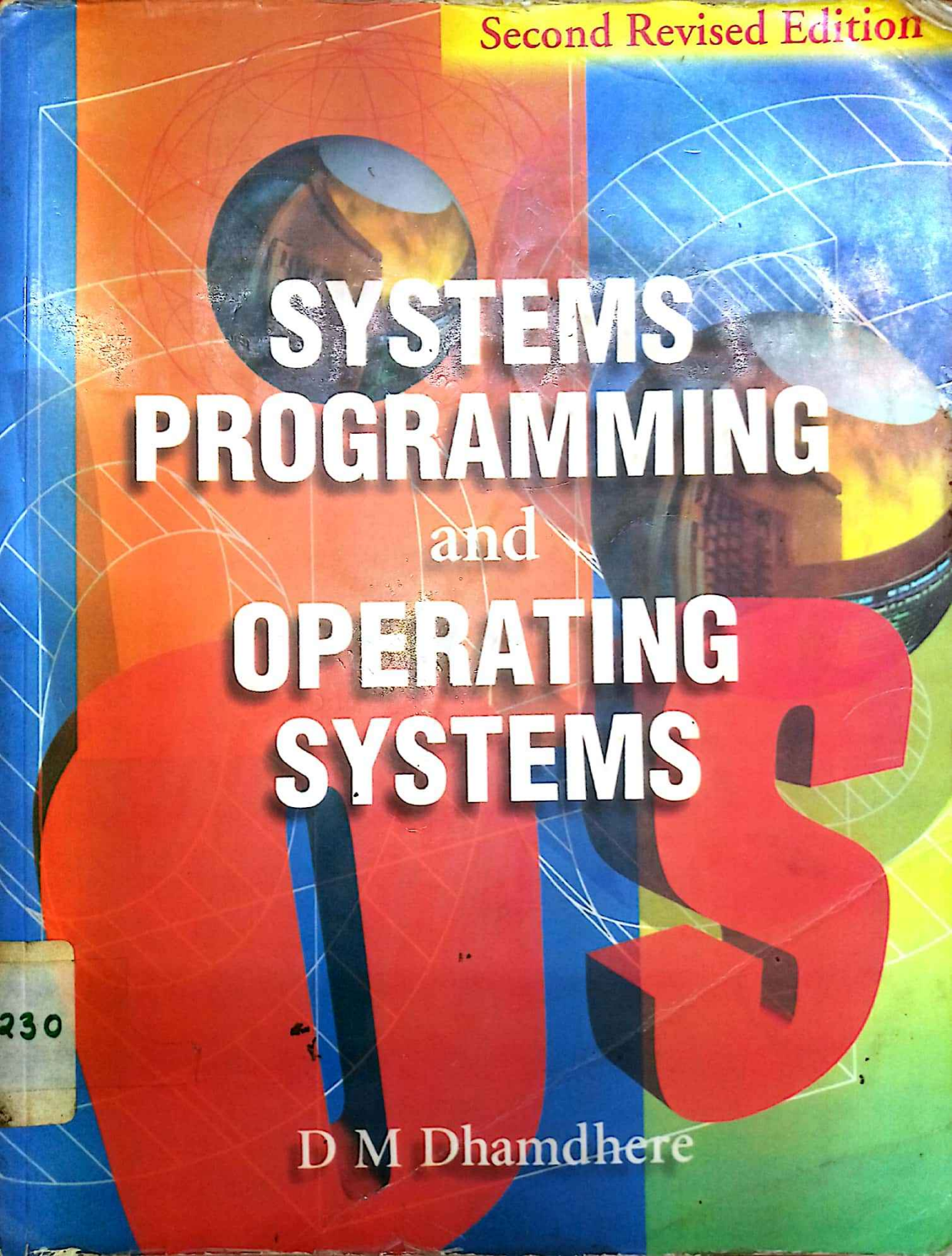
Second Revised Edition



SYSTEMS PROGRAMMING and OPERATING SYSTEMS

D M Dhamdhere

Systems Programming
and
Operating Systems

Second Revised Edition

Systems Programming and Operating Systems

Second Revised Edition

D M Dhamdhere

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

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

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

Tata McGraw-Hill

34th reprint 2010
RBXCRRZZRALRC

ISBN-13: 978-0-07-463579-7
ISBN-10: 0-07-463579-4

Published by Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008, and printed at
India Binding House, Noida 201 301

Dedicated to

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

Preface to the Second Revised Edition

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

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

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

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

D M DHAMDHERE

Systems Programming
and
Operating Systems

Second Revised Edition

Preface to the First Edition

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

Organization of the book

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

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

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

Using this book

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

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

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

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

DM DHAMDHERE

Contents

Preface to the Second Edition

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

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

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

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

D M DHAMDHERE

Part I

SYSTEMS PROGRAMMING

CHAPTER 1

Language Processors

1.1 INTRODUCTION

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

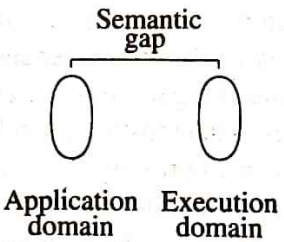


Fig. 1.1 Semantic gap

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

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

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

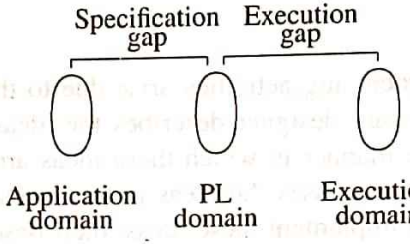


Fig. 1.2 Specification and execution gaps

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

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

Language processors

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

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

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

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

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

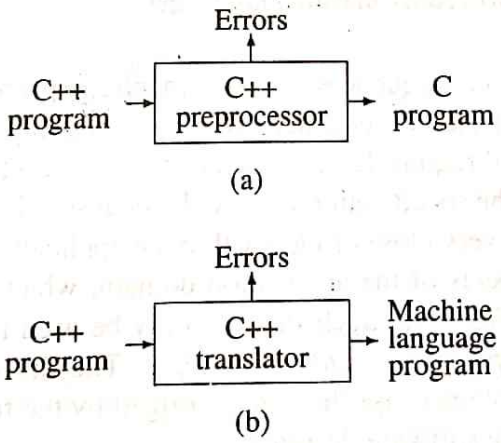


Fig. 1.3 Language processors

Interpreters

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

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

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

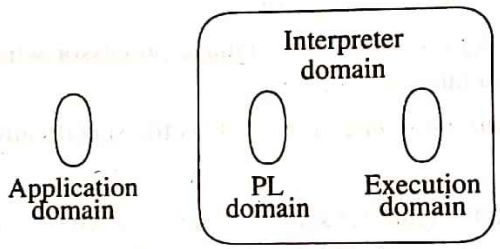


Fig. 1.4 Interpreter

Problem oriented and procedure oriented languages

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

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

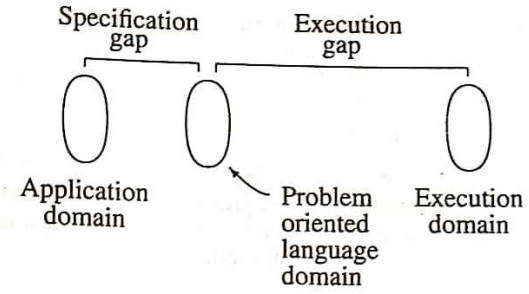


Fig. 1.5 Problem oriented language domain

1.2 LANGUAGE PROCESSING ACTIVITIES

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

1. Program generation activities
2. Program execution activities.

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

1.2.1 Program Generation

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

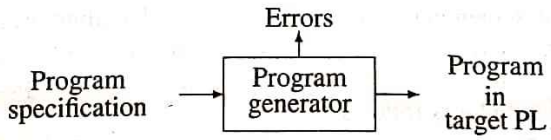


Fig. 1.6 Program generation

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

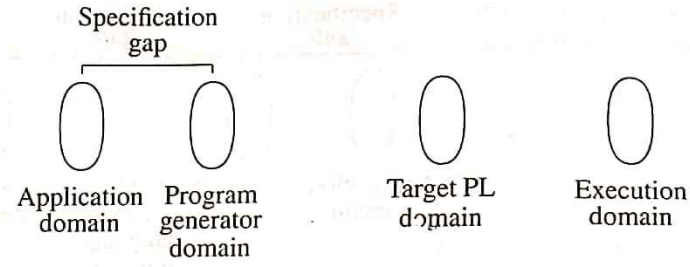


Fig. 1.7 Program generator domain

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

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

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

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

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

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

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

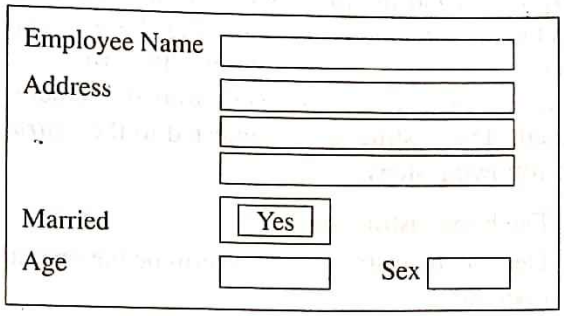


Fig. 1.8 Screen displayed by a screen handling program

1.2.2 Program Execution

Two popular models for program execution are translation and interpretation.

Program translation

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

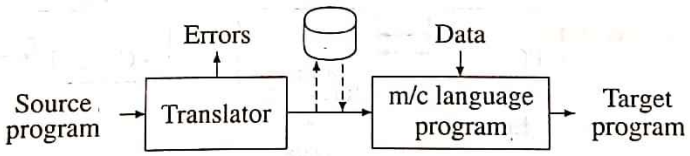


Fig. 1.9 Program translation model

Characteristics of the program translation model are:

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

Program interpretation

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

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

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

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

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

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

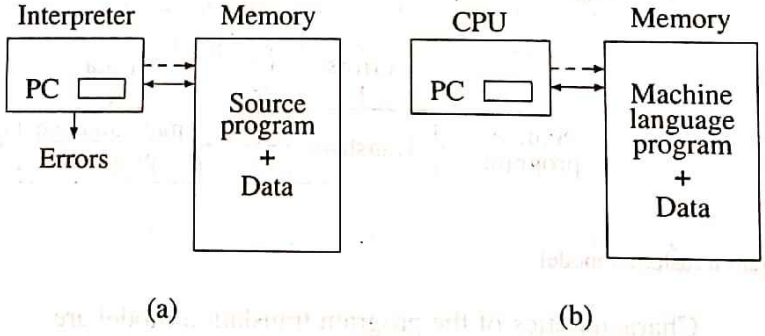


Fig. 1.10 Schematics of (a) interpretation, (b) program execution

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

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

Section 6.6 contains a detailed description of interpretation.

Comparison

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

1.3 FUNDAMENTALS OF LANGUAGE PROCESSING

Definition 1.2 (Language Processing)

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

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

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

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

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

Example 1.3 Consider the statement

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

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

$$\frac{\texttt{profit} \times 100}{\texttt{cost\_price}}$$

to `percent_profit`.

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

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

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

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

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

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

Phases and passes of a language processor

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

- Forward references

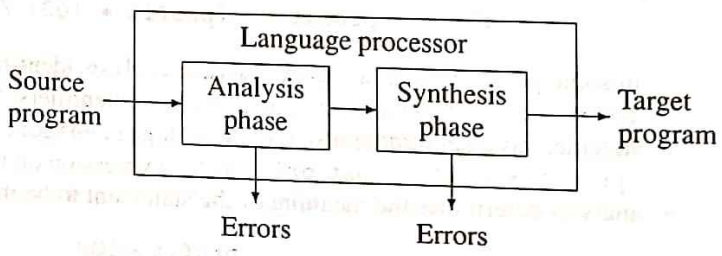


Fig. 1.11 Phases of a language processor

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

We discuss these issues in the following.

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

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

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

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

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

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

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

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

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

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

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

Intermediate representation of programs

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

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

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

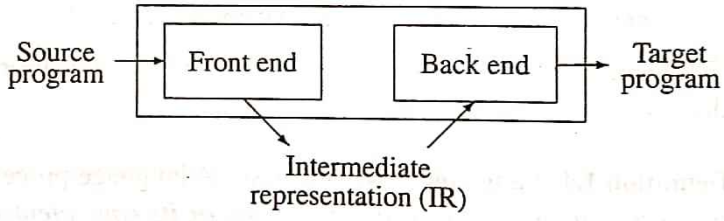


Fig. 1.12 Two pass schematic for language processing

Desirable properties of an IR are:

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

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

Semantic actions

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

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

1.3.1 A Toy Compiler

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

1.3.1.1 The Front End

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

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

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

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

Output of the front end

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

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

Tables

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

Intermediate code (IC)

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

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

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

Symbol table

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

Intermediate code

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

Fig. 1.13 IR for the program of Example 1.8

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

Lexical analysis (Scanning)

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

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

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

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

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

Syntax analysis (Parsing)

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

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

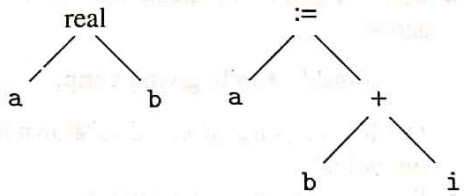


Fig. 1.14 IC for the statements `a,b : real; a:=b+i;`

Semantic analysis

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

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

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

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

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

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

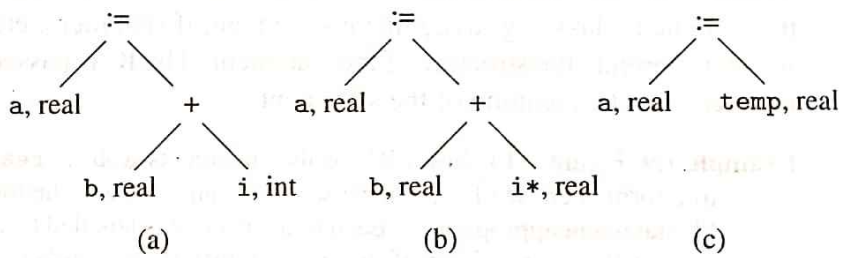


Fig. 1.15 Steps in semantic analysis of an assignment statement

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

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

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

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

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

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

1.3.1.2 The Back End

The back end performs memory allocation and code generation.

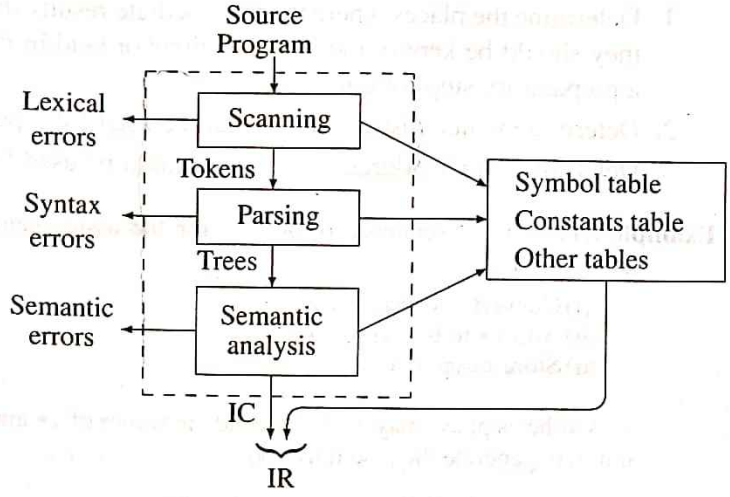


Fig. 1.16 Front end of the toy compiler

Memory allocation

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

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

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

Fig. 1.17 Symbol table after memory allocation

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

Code generation

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

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

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

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

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

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

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

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

Figure 1.18 shows a schematic of the back end.

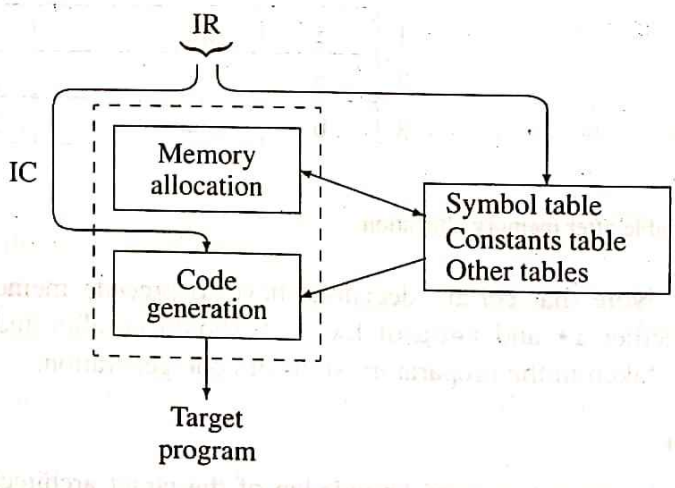


Fig. 1.18 Back end of the toy compiler

1.4 FUNDAMENTALS OF LANGUAGE SPECIFICATION

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

1.4.1 Programming Language Grammars

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

Terminal symbols, alphabet and strings

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

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

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

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

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

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

Nonterminal symbols

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

Productions

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

A nonterminal symbol :: = String of Ts and NTs

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

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

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

$$<\textit{Noun Phrase}> \quad ::= \quad <\textit{Article}><\textit{Noun}>$$

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

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

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

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

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

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

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

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

Derivation, reduction and parse trees

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

Derivation

Let production P_1 of grammar G be of the form

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

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

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

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

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

Example 1.15 Consider the grammar G

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

The following strings are sentential forms of L_G:

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

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

Reduction

Let production P_1 of grammar G be of the form

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

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

Example 1.16 To determine the validity of the string

the boy ate an apple

according to grammar (1.2) we perform the following reductions

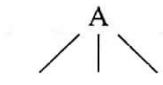
Step	*String*
0	the boy ate an apple
1	< *Article* > boy ate an apple
2	< *Article* > < *Noun* > ate an apple
3	< *Article* > < *Noun* > < *Verb* > an apple
4	< *Article* > < *Noun* > < *Verb* > < *Article* > apple
5	< *Article* > < *Noun* > < *Verb* > < *Article* > < *Noun* >
6	< *Noun Phrase* > < *Verb*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

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

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

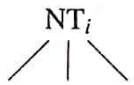
Parse trees

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



(Sequence of Ts and NTs constituting α)

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



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

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

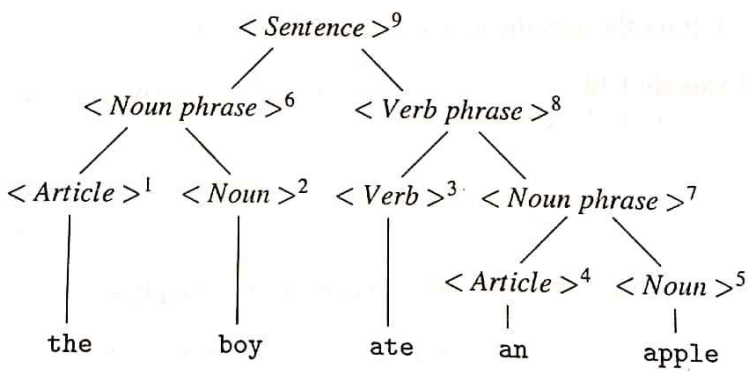


Fig. 1.19 Parse tree

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

Recursive specification

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

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

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

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

$$
\begin{aligned}
<id> &::= <letter> \mid <id><letter> \mid <id><digit> \\
<const> &::= <digit> + <digit> \mid - <digit> \\
&\quad \mid <const><digit>
\end{aligned}
$$

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

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

$$
\begin{aligned}
<exp> &::= <term> \mid <term> + <term> \\
&\quad \mid <term> + <term> + <term> \mid \ldots
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

1.4.1.1 Classification of Grammars

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

Type-0 grammars

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

$$\alpha ::= \beta$$

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

Type-1 grammars

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

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

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

Type-2 grammars

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

$$A ::= \pi$$

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

Type-3 grammars

Type-3 grammars are characterized by productions of the form

A ::= t B | t or
A ::= B t | t

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

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

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

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

Operator grammars

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

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

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

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

1.4.1.2 Ambiguity in Grammatic Specification

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

Example 1.20 Consider the expression grammar

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

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

Eliminating ambiguity

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

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

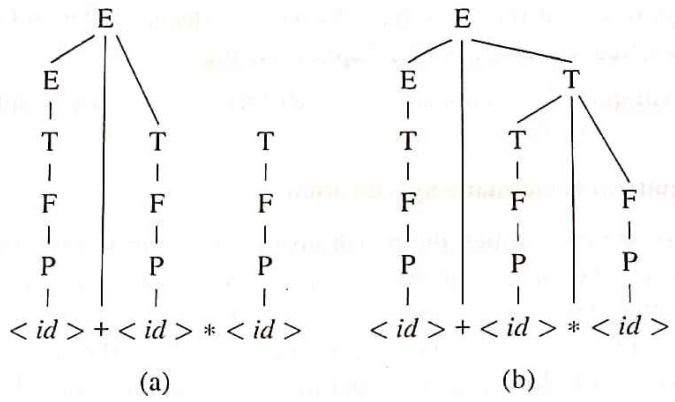


Fig. 1.20 Ensuring a unique parse tree for an expression

EXERCISE 1.4

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

1.4.2 Binding and Binding Times

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

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

```
var : type;
```

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

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

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

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

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

Example 1.22 Consider the Pascal program

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

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

Importance of binding times

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

Example 1.23 Consider the PL/1 program segment

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

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

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

Static and dynamic bindings

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

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

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

1.5 LANGUAGE PROCESSOR DEVELOPMENT TOOLS

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

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

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

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

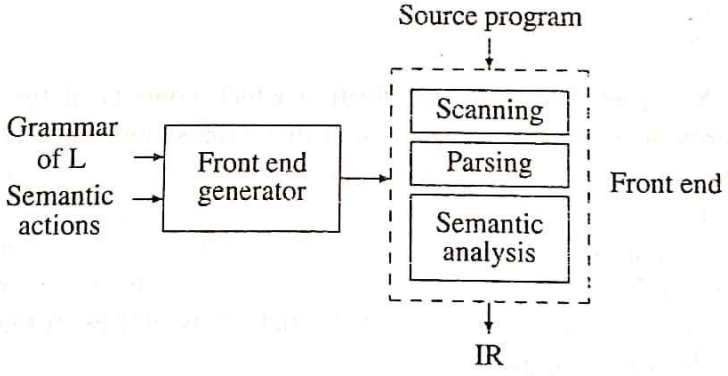


Fig. 1.21 A language processor development tool (LPDT)

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

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

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

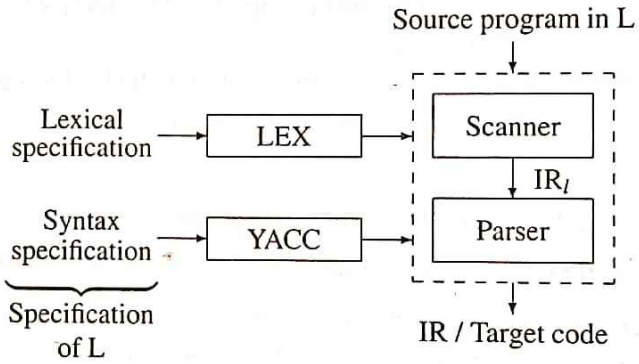


Fig. 1.22 Using LEX and YACC

1.5.1 LEX

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

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

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

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

Fig. 1.23 A sample LEX specification

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

1.5.2 YACC

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

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

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

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

Fig. 1.24 A sample YACC specification

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

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

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

BIBLIOGRAPHY

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

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

(a) Programming languages

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

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

(b) Grammars for programming languages

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

(c) Program generation

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

(d) Language processor development tools

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