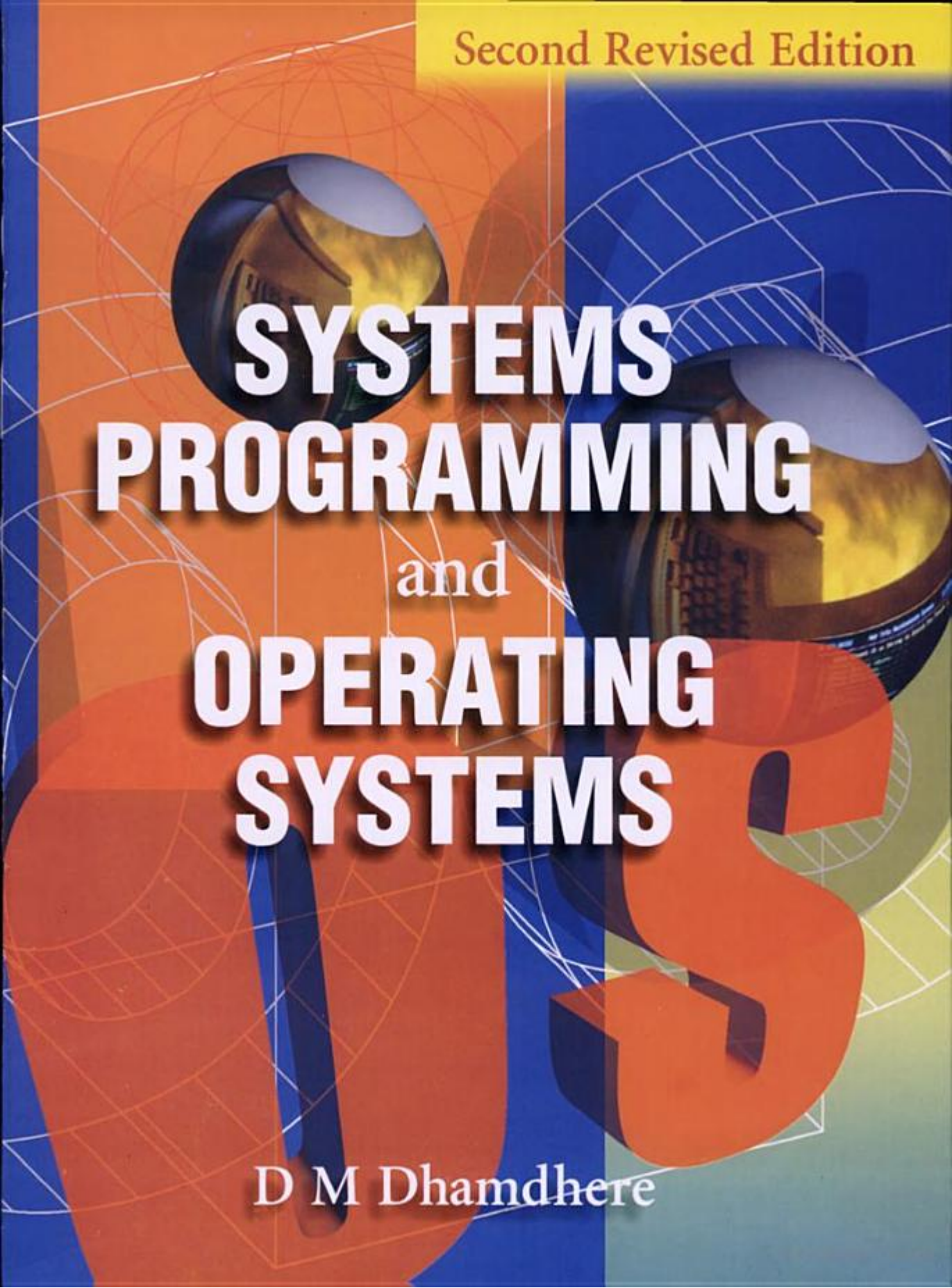
Second Revised Edition



SYSTEMS PROGRAMMING and OPERATING SYSTEMS

D M Dhamdhere

Tata McGraw-Hill

30th reprint 2009
RQXLCDDXRQRYL

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

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

The McGraw-Hill Companies

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.

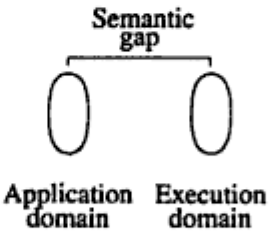


Fig. 1.1 Semantic gap

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.

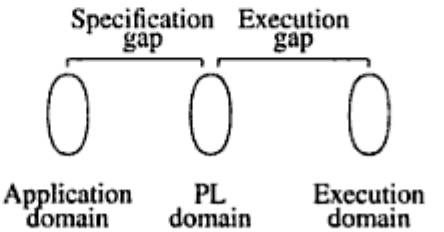


Fig. 1.2 Specification and execution gaps

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.

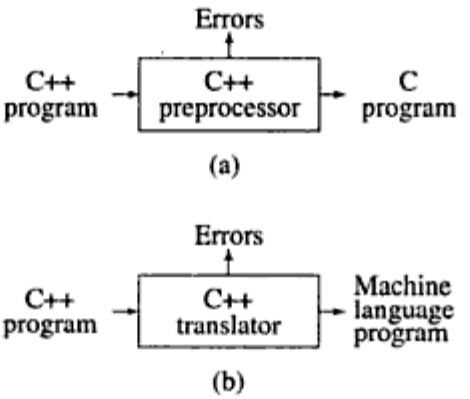


Fig. 1.3 Language processors

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.

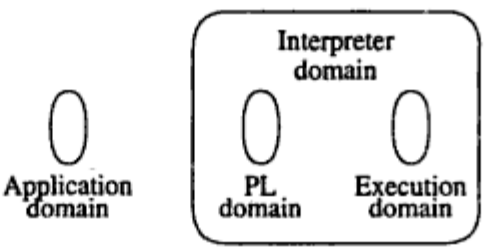


Fig. 1.4 Interpreter

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.

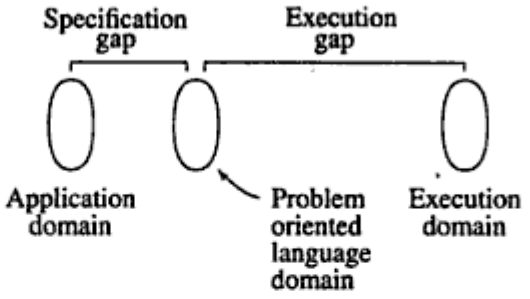


Fig. 1.5 Problem oriented language domain

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.

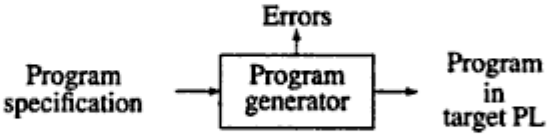


Fig. 1.6 Program generation

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.

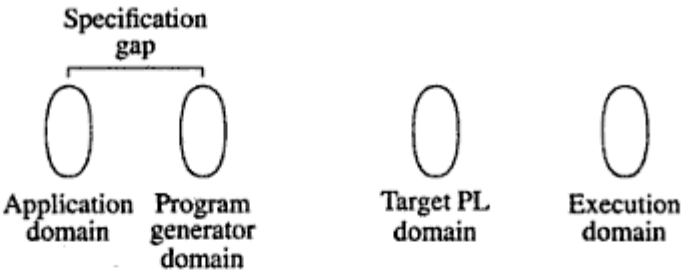


Fig. 1.7 Program generator domain

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.

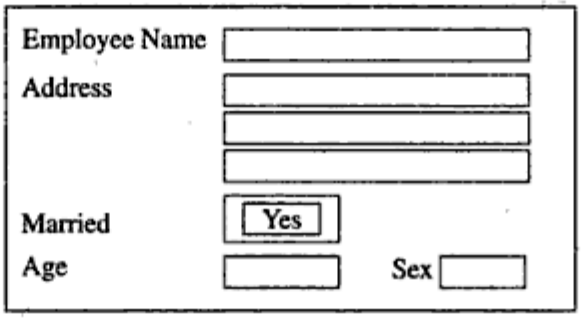


Fig. 1.8 Screen displayed by a screen handling program

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).

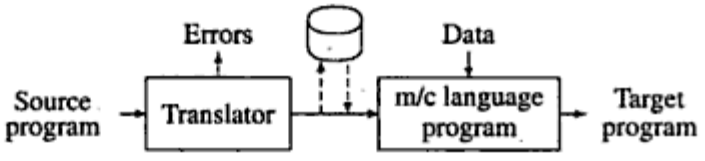


Fig. 1.9 Program translation model

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.

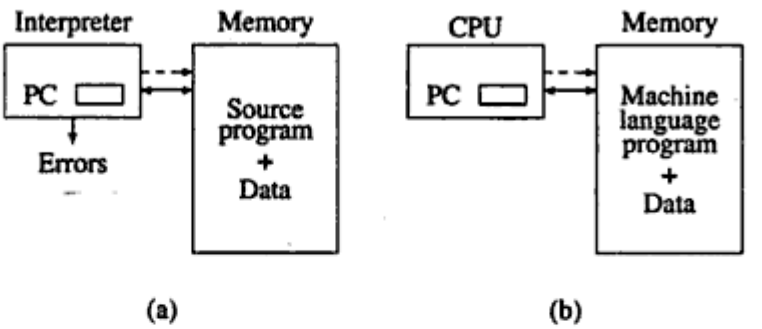


Fig. 1.10 Schematics of (a) interpretation, (b) program execution

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

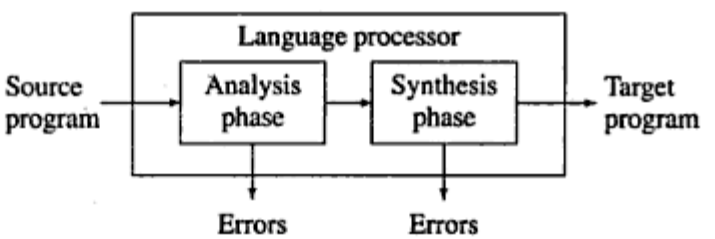


Fig. 1.11 Phases of a language processor

- 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.

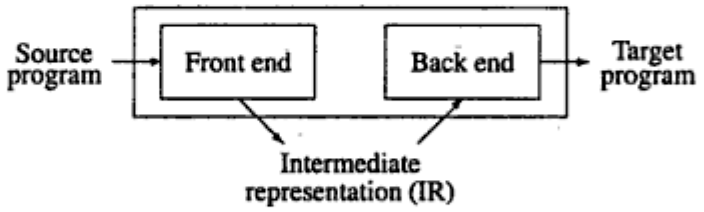


Fig. 1.12 Two pass schematic for language processing

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.

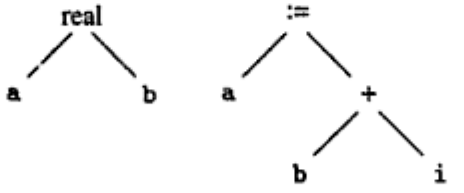


Fig. 1.14 IC for the statements `a,b : real; a := b+i;`

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.

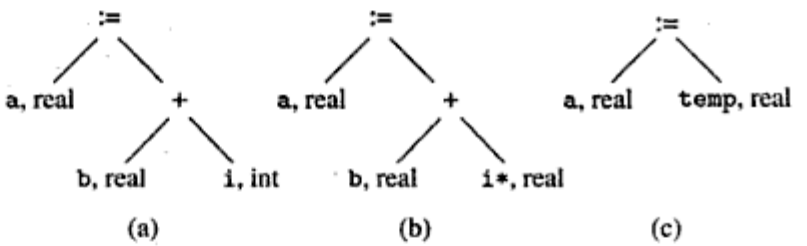


Fig. 1.15 Steps in semantic analysis of an assignment statement

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.

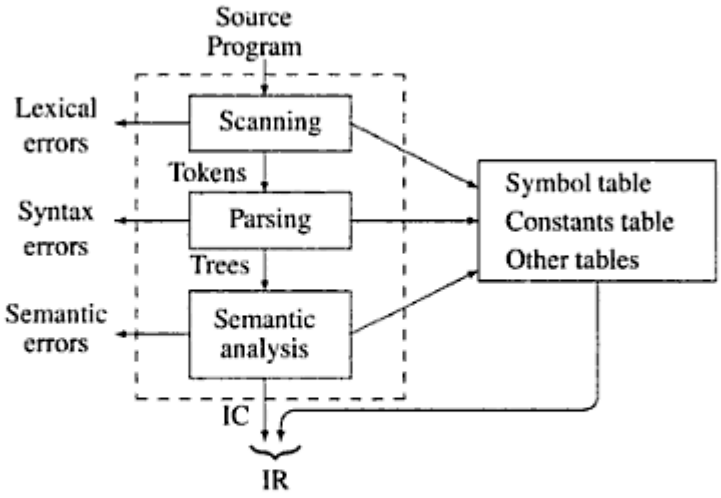


Fig. 1.16 Front end of the toy compiler

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.

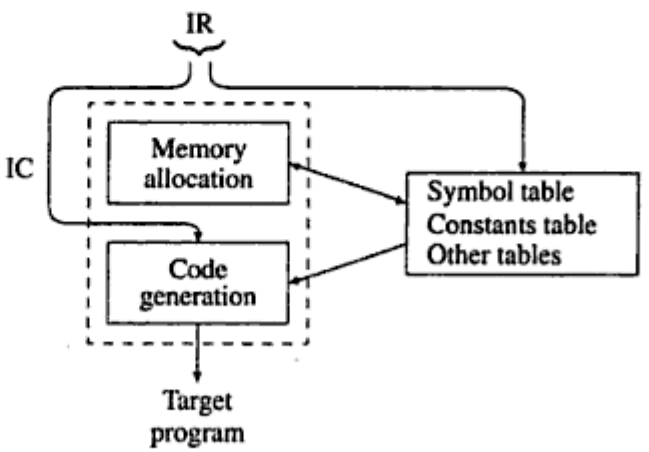


Fig. 1.18 Back end of the toy compiler

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 < ... >, 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 $\eta \overset{*}{\rightarrow} N$ to depict direct and transitive reduction, respectively. Thus, $\alpha \rightarrow A$ only if $A ::= \alpha$ is a production of G and $\alpha \overset{*}{\rightarrow} A$ 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 $\delta \overset{*}{\rightarrow} S$, 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

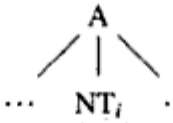
Step	*String*
0	`the boy ate an apple`
1	< *Article* > `boy ate an apple`
2	< *Article* > < *Noun* > `ate an apple`
3	< *Article* > < *Noun* > < *Verb* > `an apple`
4	< *Article* > < *Noun* > < *Verb* > < *Article* > `apple`
5	< *Article* > < *Noun* > < *Verb* > < *Article* > < *Noun* >
6	< *Noun Phrase* > < *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` $\overset{*}{\rightarrow}$ < *Sentence* >.

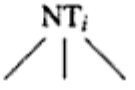
Parse trees

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* >.

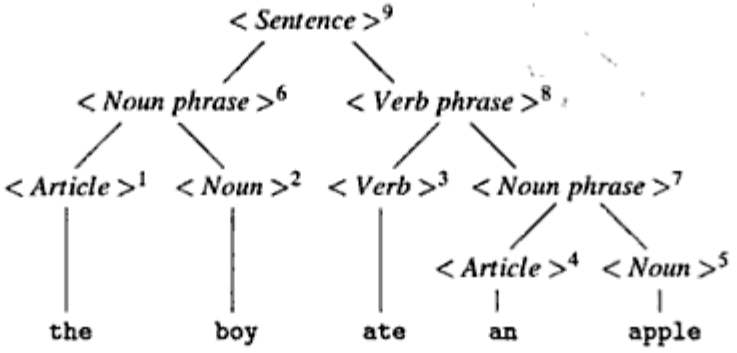


Fig. 1.19 Parse tree

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 ↑ (exponentiation), * and +.

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

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

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

$$
\begin{aligned}
\langle id \rangle &::= \langle letter \rangle | \langle id \rangle \langle letter \rangle | \langle id \rangle \langle digit \rangle \\
\langle const \rangle &::= \langle digit \rangle + \langle digit \rangle | - \langle digit \rangle \\
&\quad | \langle const \rangle \langle digit \rangle
\end{aligned}
$$

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

$$
\begin{aligned}
\langle exp \rangle &::= \langle term \rangle | \langle term \rangle + \langle term \rangle \\
&\quad | \langle term \rangle + \langle term \rangle + \langle term \rangle | \ldots
\end{aligned}
$$

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

$$
\langle exp \rangle ::= \langle exp \rangle + \langle term \rangle | \langle term \rangle \tag{1.4}
$$

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 charaeters, 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

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

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

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

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

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

Operator grammars

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

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

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

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

1.4.1.2 Ambiguity in Grammatic Specification

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

Example 1.20 Consider the expression grammar

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

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

Eliminating ambiguity

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

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

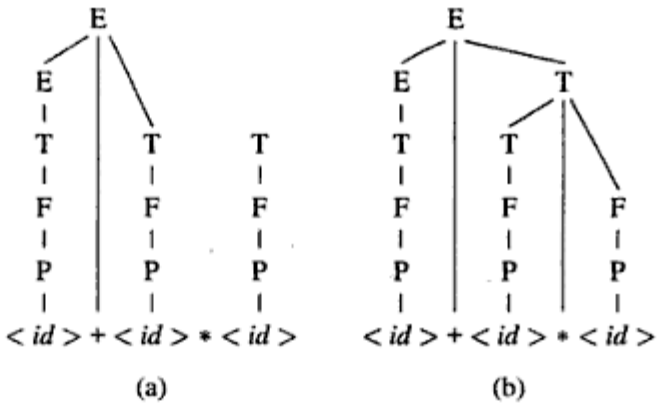


Fig. 1.20 Ensuring a unique parse tree for an expression

EXERCISE 1.4

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

1.4.2 Binding and Binding Times

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

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

```
var : type;
```

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

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

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

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

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

Example 1.22 Consider the Pascal program

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

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

Importance of binding times

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

Example 1.23 Consider the PL/1 program segment

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

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

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

Static and dynamic bindings

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

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

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

1.5 LANGUAGE PROCESSOR DEVELOPMENT TOOLS

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

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

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

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

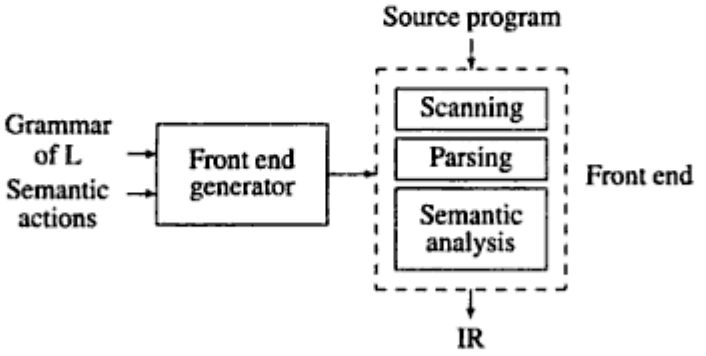


Fig. 1.21 A language processor development tool (LPDT)

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

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

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.

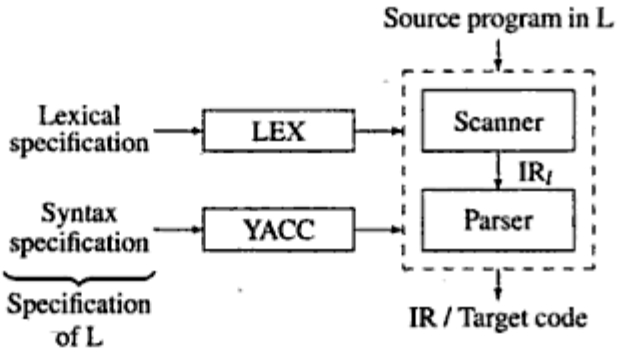


Fig. 1.22 Using LEX and YACC

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*, 2nd edition, O'Reilly & associates, Sebastopol.

CHAPTER 2

Data Structures for Language Processing

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

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

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

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

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

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

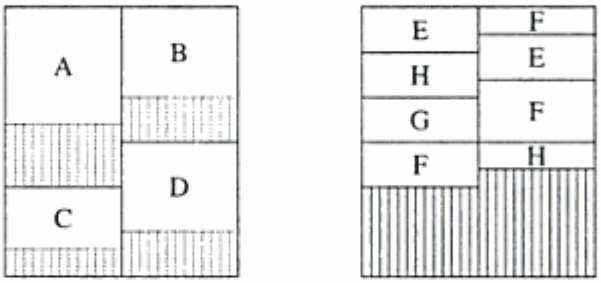


Fig. 2.1 Linear and nonlinear data structures

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

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

Example 2.2 Consider the Pascal program

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

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

Example 2.3 Consider the following Pascal and C program segments

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

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

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

2.1 SEARCH DATA STRUCTURES

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

Entry formats

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

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

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

Variant part:

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

Fixed and variable length entries

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

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

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

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

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

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

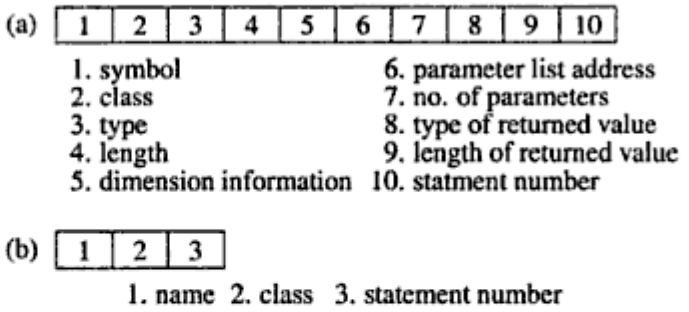


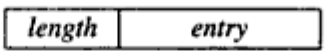
Fig. 2.2 (a) Fixed entry, (b) Variable length entry for label

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

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

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

We will depict this format as



Hybrid entry formats

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

Operations on search structures

The following operations are performed on search data structures:

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

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

Generic search procedure

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

Algorithm 2.1 (Generic search procedure)

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

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

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

2.1.1 Table Organizations

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

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

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

Sequential search organization

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

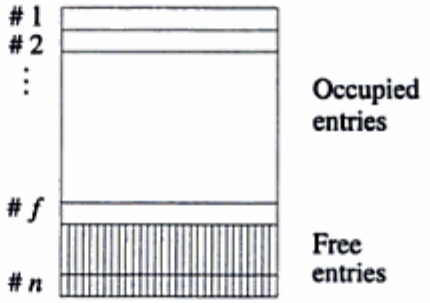


Fig. 2.3 Sequential search table

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

Search for a symbol

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

$p_s = f/2$ for a successful search
$p_u = f$ for an unsuccessful search

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

Add a symbol

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

Delete a symbol

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

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

Active/deleted	*Symbol*	*Other info*

Binary search organization

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

Algorithm 2.2 (Binary search)

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

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

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

Hash table organization

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

Algorithm 2.3 (Hash table management)

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

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

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

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

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

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

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

Hashing functions

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

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

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

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

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

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

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

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

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

Collision handling methods

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

Rehashing

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

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

to provide a series of hashing functions for rehashing.

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

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

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

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

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

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

Table 2.1 Performance of sequential rehash

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

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

Overflow chaining

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

Symbol	*Other info*	*Pointer*

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

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

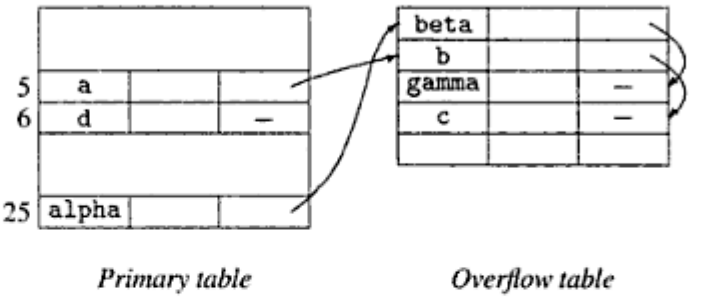


Fig. 2.4 Hash table organization with overflow chaining

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

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

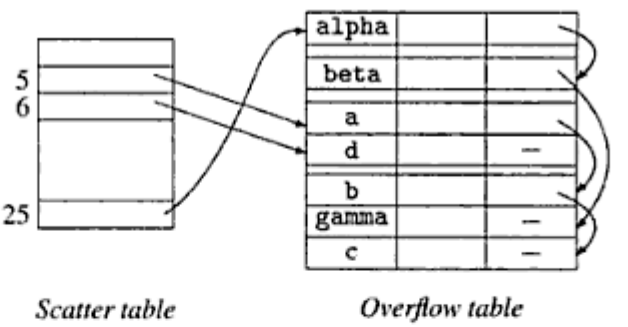


Fig. 2.5 Scatter table organization

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

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

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

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

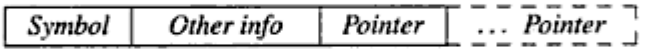
Memory requirements in the case of hash with overflow chaining (see Ex. 2.7) would have been

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

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

2.1.2 Linked List and Tree Structured Organizations

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



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

Linked lists

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

Binary trees

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

Algorithm 2.4 (Binary tree search)

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

2. If $s = (current\_node\_pointer)^*.symbol$, then exit with success;
3. If $s < (current\_node\_pointer)^*.symbol$ then
   current_node_pointer := $(current\_node\_pointer)^*.left\_pointer$;
   else *current_node_pointer* := $(current\_node\_pointer)^*.right\_pointer$;
4. If *current_node_pointer* = *nil* then
   exit with failure.
   else goto Step 2.

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

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

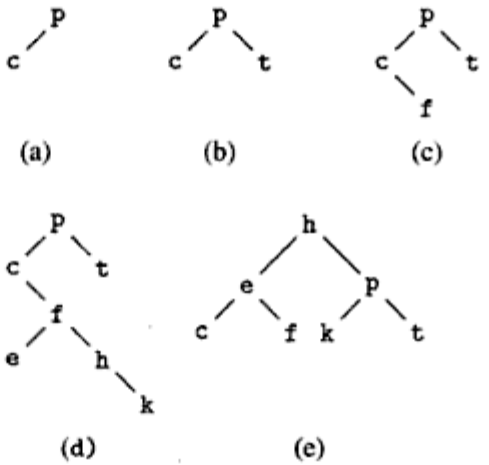


Fig. 2.6 Tree structured table organization

Nested search structures

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

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

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

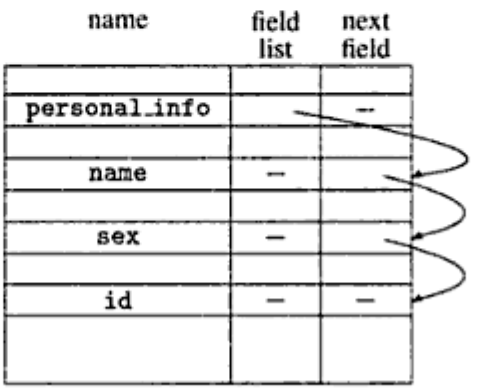


Fig. 2.7 Multi-list structure

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

EXERCISE 2.1

1. Comment on the feasibility of designing a hashing function to implement a direct entry organization for a fixed set of symbols. (Sprugnoli (1977) and Lyon (1978) report some related work).
2. An assembler is to be designed to handle programs containing up to 500 symbols. However, it is found that an average program contains only about 200 symbols. It is decided that a hash organization with sequential rehashing must be used. Evaluate the following three design alternatives in terms of memory requirements and access efficiency.
   (a) Size of the table = 550 entries
   (b) Size of the table = 700 entries
   (c) Size of the table = 500 entries
3. An assembler supports an option by which it produces an alphabetical listing of all symbols used in a program. Comment on the suitability of the following organizations

to organize the symbol table:

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

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

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

   Comment on the advantages of this organization.
5. Discuss the problem of deletion of entries in the following symbol table organizations:

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

2.2 ALLOCATION DATA STRUCTURES

We discuss two allocation data structures, stacks and heaps.

2.2.1 Stacks

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

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

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

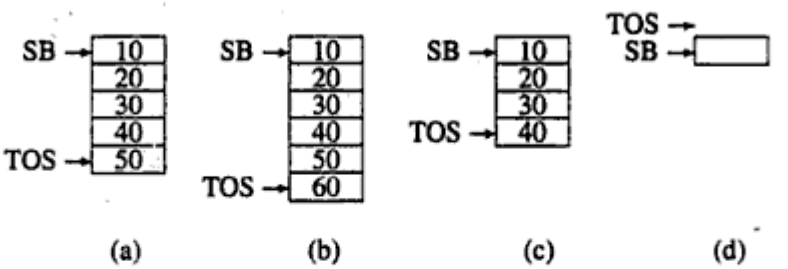


Fig. 2.8 Stack model of allocation

Extended stack model

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

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

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

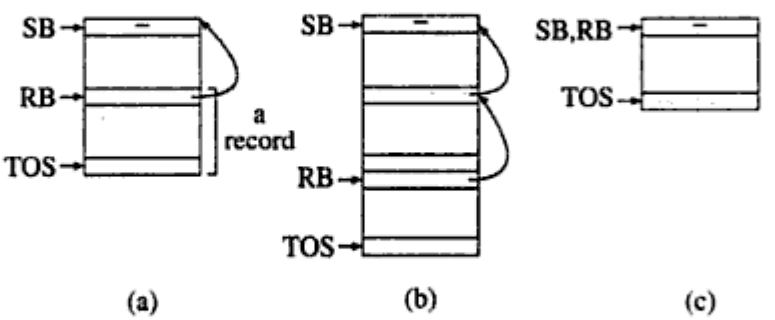


Fig. 2.9 Extended stack model

Allocation time actions

No.	Statement
1.	TOS := TOS + 1;
2.	TOS* := RB;
3.	RB := TOS;
4.	TOS := TOS + n;

The first statement increments TOS by one stack entry. It now points at the *reserved pointer* of the new record. The '*' mark in statement 2 indicates indirection. Hence the assignment TOS* := RB deposits the address of the previous record base into the reserved pointer. Statement 3 sets RB to point at the first stack entry in the new record. Statement 4 performs allocation of n stack entries to the new entity (see Fig. 2.9(b)). The newly created entity now occupies the addresses <RB> +l to <RB> +$l \times n$, where <RB> stands for 'contents of RB'.

Deallocation time actions

No.	Statement
1.	TOS := RB − 1;
2.	RB := RB*;

The first statement pops a record off the stack by resetting TOS to the value it had before the record was allocated. RB is then made to point at the base of the previous record (see Fig. 2.9(c)).

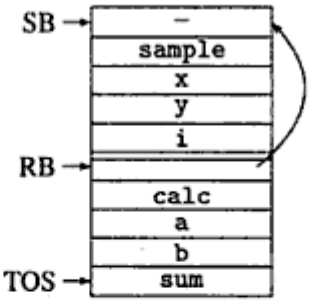


Fig. 2.10 Stack structured symbol table

Example 2.11 When a Pascal program contains nested procedures, many symbol tables must co-exist during compilation. Figure 2.10 shows the symbol tables of the main program and procedure `calc` of the Pascal program of Ex. 2.2 when the statement `sum := a+b` is being compiled. Note the address contained in the *reserved pointer* in the symbol table for procedure `calc`. It is used to pop the symbol table of `calc` off the stack after its `end` statement is compiled.

2.2.2 Heaps

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

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

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

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

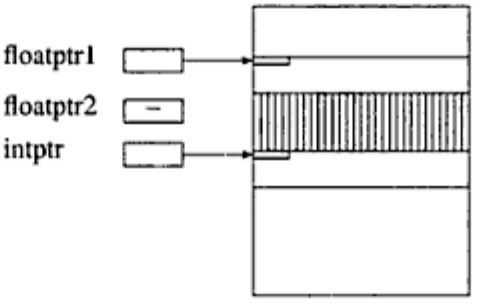


Fig. 2.11 Heap

Memory management

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

Identifying free memory areas

Two popular techniques used to identify free memory areas are:

1. Reference counts
2. Garbage collection.

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

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

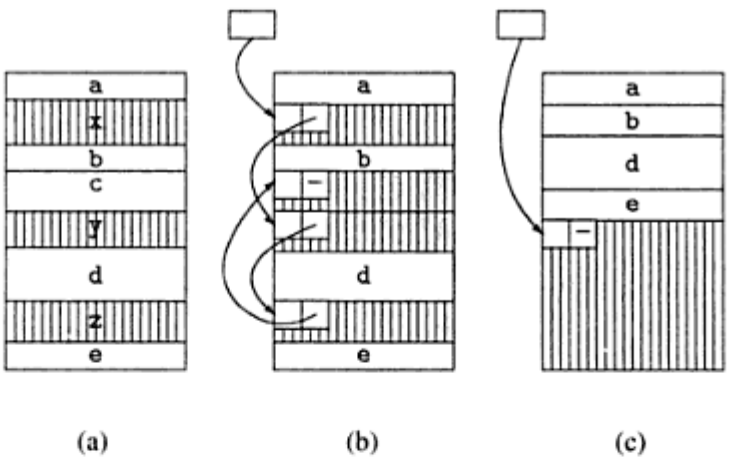


Fig. 2.12 (a) allocation status of heap, (b) free list, (c) after compaction

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

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

Reuse of memory

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

1. First fit technique
2. Best fit technique.

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

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

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

BIBLIOGRAPHY

(a) Data structures, general

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

(b) Table management

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

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

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

CHAPTER 3

Scanning & Parsing

3.1 SCANNING

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

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

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

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

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

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

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

Finite state automata

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

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

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

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

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

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

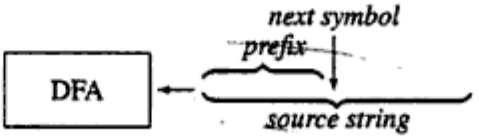


Fig. 3.1 Operation of a DFA

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

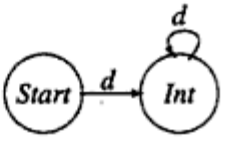
The validity of a string is determined by giving it at the input of a DFA in its initial state. The string is valid if and only if the DFA recognizes every symbol in the string and finds itself in a final state at the end of the string. This fact follows from the deterministic nature of transitions in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

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

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

	Next Symbol
State	d
Start	Int
Int	Int

(a)



(b)

Fig. 3.2 DFA to recognise integer strings

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

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

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

Regular expressions

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

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

$$\begin{aligned} &\langle \textit{section code} \rangle &::= \quad & l \mid \langle \textit{section code} \rangle\, l \\ &\langle \textit{numeric code} \rangle &::= \quad & d \mid \langle \textit{numeric code} \rangle\, d \\ &\langle \textit{employee code} \rangle &::= \quad & \langle \textit{section code} \rangle\, \langle \textit{numeric code} \rangle \end{aligned}$$

Note that a specification like

$$\langle s\_code \rangle \quad ::= \quad l \mid d \mid \langle s\_code \rangle\, l \mid \langle s\_code \rangle\, d$$

would be incorrect !

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

Table 3.1 Regular expressions

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

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

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

Some other examples of regular expressions are

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

Building DFAs

A DFA for a Type-3 specification can be built using some simple rules. Building a DFA for a regular expression can be achieved by repeated application of the same simple rules. However it is a tedious process, hence it is best to automate the process of building DFAs. In the following, we shall not discuss the building of DFAs but assume that they can be built by a procedure described in the literature cited at the end of the chapter.

The lexical components of a source language can be specified by a set of regular expressions. Since an input string may contain any one of these lexical components, it is necessary to use a single DFA as a recognizer for valid lexical strings in the language. Such a DFA would have a single initial state and one or more final states for each lexical component.

Example 3.5 Figure 3.3 shows a DFA for recognizing identifiers, unsigned integers and unsigned real numbers with fractions. The DFA has 3 final states – *Id*, *Int* and *Real* corresponding to identifier, unsigned integer and unsigned real respectively. Note that a string like '25.' is invalid because it leaves the DFA in state s_2 which is not a final state.

State	*Next Symbol*		
	l	*d*	.
Start	*Id*	*Int*	
Id	*Id*	*Id*	
Int		*Int*	s_2
s_2		*Real*	
Real		*Real*	

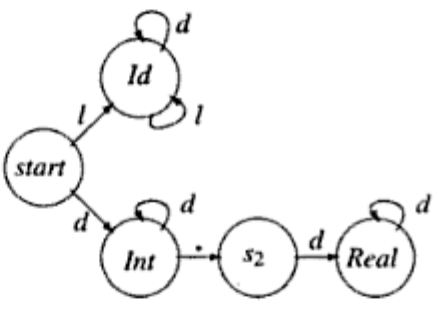


Fig. 3.3 A combined DFA for integers, real numbers and identifiers

Performing semantic actions

Semantic actions during scanning concern table building and construction of tokens for lexical components. These actions are associated with the final states of a DFA. The semantic actions associated with a final state s_f are performed after the DFA recognizes the longest valid prefix of the source string corresponding to s_f.

Writing a scanner

We will use a notation analogous to the LEX notation (see Section 1.5.1) to specify a scanner. A scanner for integer and real numbers, identifiers and reserved words of a language is given in Table 3.1.

Table 3.2 Specification of a scanner

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

EXERCISE 3.1

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

3.2 PARSING

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

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

Parse trees and abstract syntax trees

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

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

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

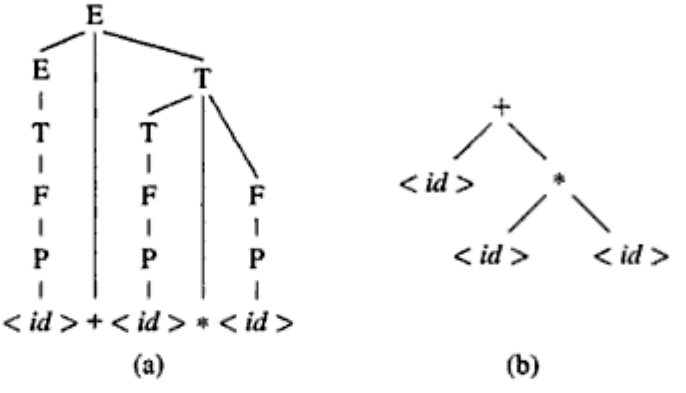


Fig. 3.4 Parse tree and AST

3.2.1 Top Down Parsing

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

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

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

Algorithm 3.1 (Naive top down parsing)

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

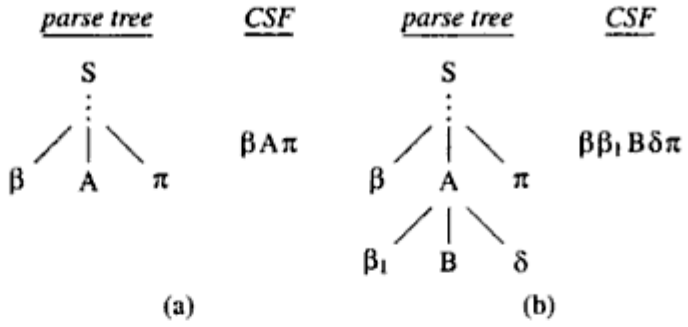


Fig. 3.5 Derivation $A \Rightarrow \beta_1 B\delta$ in top down parsing

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

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

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

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

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

Predictions and backtracking

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

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

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

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

Implementing top down parsing

The following features are needed to implement top down parsing:

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

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

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

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

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

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

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

Table 3.3 Predictions in top sown parsing

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

Comments on top down parsing

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

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

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

The first prediction would be

$$\mathrm{E} \Rightarrow \mathrm{E} + \mathrm{T}$$

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

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

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

$$\mathrm{E} ::= \mathrm{E} + T \mid \mathrm{T}$$

as

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

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

Top down parsing without backtracking

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

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

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

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

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

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

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

The complete rewritten form of Grammar (3.2) is

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

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

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

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

Table 3.4 Top down parsing without backtracking

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

3.2.1.1 Practical Top Down Parsing

A recursive descent parser

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

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

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

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

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

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

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

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

Fig. 3.6 A recursive descent parser

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

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

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

existed in grammar (3.5).

An LL(1) parser

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

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

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

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

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

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

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

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

Fig. 3.7 LL(1) parser table

Table 3.5 LL parsing

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

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

EXERCISE 3.2.1

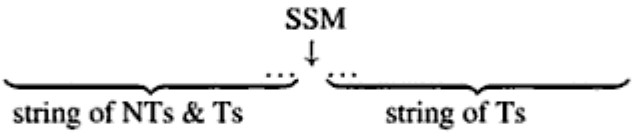
1. Study the operation of the recursive descent parser described in this section for the following input strings:
   (a) < *id* > + < *id* > * < *id* >
   (b) < *id* > * < *id* >< *id* >
   Verify that precise error indication is provided by the parser.
2. A recursive descent parser is to be developed for Grammar (1.3).

(a) What changes will you make in the grammar to make it suitable for recursive descent parsing?
(b) Code the parser procedures.
(c) Identify the procedures that perform error indication.

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

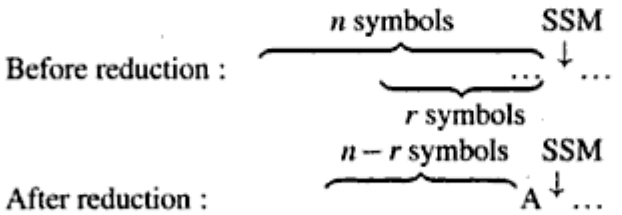
3.2.2 Bottom Up Parsing

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



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

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



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

Algorithm 3.2 (Naive bottom up parsing)

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

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

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

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

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

SSM
↓
... αtγ ...

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

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

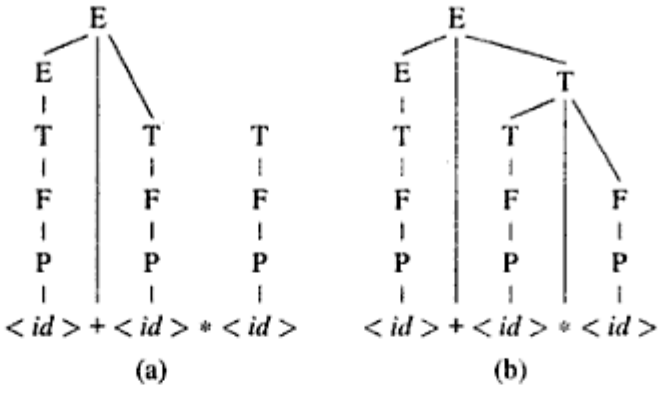


Fig. 3.8 Parse trees in bottom up parsing

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

Simple precedence

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

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

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

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

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

(3.7)

Definition 3.4 (Simple precedence grammar (SPG))

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

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

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

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

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

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

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

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

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

Algorithm 3.3 (Bottom up parsing)

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

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

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

Operator precedence grammars

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

Operator precedence

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

1. $a \doteq b$ iff there exists a grammar production
   $C ::= \beta ab\gamma$ or $C ::= \beta aAb\gamma$.
2. $a \cdot> b$ iff there exist grammar productions
   $C ::= \beta Ab\gamma$ and
   $A ::= \pi a \mid \pi aD$
3. $a <\cdot b$ iff there exist grammar productions
   $C ::= \beta aB\gamma$ and
   $B ::= b\delta \mid Db\delta$

(3.8)

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

Example 3.13 The precedence relations for the grammar

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

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

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

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

Fig. 3.9

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

Example 3.14 Consider the following grammar

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

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

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

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

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

Operator precedence parsing

Example 3.15 Consider parsing of the string

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

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

Fig. 3.10 Operator precedence matrix

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

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

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

Reduced string : ⊢ <*id*> + ... ⊣
<· ·>

which leads to reduction of '+'.

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

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

Algorithm 3.4 (Operator Precedence Parsing)
Data structures

Stack : Each stack entry is a record with two fields, *operator* and *operand_pointer*.
Node : A *node* is a record with three fields, *symbol*, *left_pointer*, and *right_pointer*.

Functions

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

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

Example 3.16 Consider parsing of the string

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

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

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

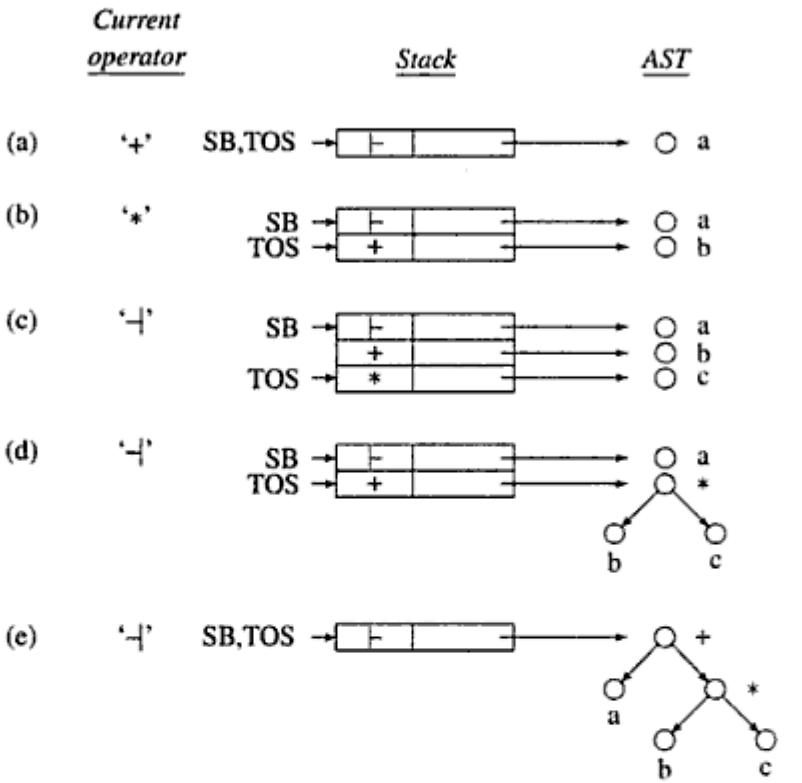


Fig. 3.11 Operator precedence parsing using a stack

LALR parsing

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

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

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

```
%%
E   :  E + T   {$$ = build_node('+', $1, $3)}
    |  Term    {$$ = $1}
    ;
T   :  T * V   {$$ = build_node('*', $1, $3)}
    |  V       {$$ = $1}
    ;
V   :  id      {$$ = build_node($1, nil, nil)}
    ;
%%
build_node(node_label, pointer_1, pointer_2);
{
     /* Build a node and return a pointer to it */
}
```

Fig. 3.12 YACC specification for an expression parser

EXERCISE 3.2

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

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

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

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

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

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

BIBLIOGRAPHY

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

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

CHAPTER 4

Assemblers

4.1 ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

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

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

Statement format

An assembly language statement has the following format:

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

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

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

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

A simple assembly language

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

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

Fig. 4.1 Mnemonic operation codes

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

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