UNIT 1
SYNTAX AND SEMANTICS

Reasons for Studying Programming Languages

Increased capacity to express ideas:

- People can easily express their ideas clearly in any language only when they have clear understanding of the natural language.
- Similarly, if programmers want to simulate the features of languages in another language, they should have some ideas regarding the concepts in other languages as well.

Improved background for choosing appropriate languages

- Many programmers when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to the project.
- If these programmers were familiar with a wider range of languages, they would be better able to choose the language that includes the features that best address the characteristics of the problem at hand.

Increased ability to learn new languages

- In software development, continuous learning is essential.
- The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only two or more languages.
- Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.

Better understanding the significance of implementation

- An understanding of implementation issues leads to an understanding of why languages are designed the way they are.
- This knowledge in turn leads to the ability to use a language more intelligently, as it was designed to use.
- We can become better programmers by understanding the choices among programming language constructs and consequences of those choices.

Better use of languages that are already known

- By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

Downloaded From EnggTree.com

Overall advancement of computing

- There is a global view of computing that can justify the study of programming language concepts.
- For example, many people believe it would have been better if ALGOL 60 had displaced Fortran in the early 1960s, because it was more elegant and had much better control statements than Fortran. That it did not is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60.
- If those who choose languages were better informed, perhaps, better languages would eventually squeeze out poorer ones.

Evolution of programming language

Zuse's Plankalkül

It was developed in 1945, its description was not published until 1972. It was never implemented

Historical Background

Between 1936 and 1945, German scientist Konrad Zuse (pronounced "Tsoozuh") putforth an effort to develop a language for expressing computations , a project he had begun in 1943 as a proposal for his Ph.D. dissertation. He named this language Plankalkül, which means program calculus. In a lengthy manuscript dated 1945 but not published until 1972, Zuse defined Plankalkül and wrote algorithms in the language to solve a wide variety of problems.

Language Overview

The simplest data type in Plankalkül was the single bit. Integer and floating-point numeric types were built from the bit type. The floating-point type used twos-complement notation

In addition to the usual scalar types, Plankalkül included arrays and records (called structs in the C-based languages). The records could include nested records.

Plankalkül included a selection statement, but it did not allow an else clause.

One of the most interesting features of Zuse's programs was the inclusion of mathematical expressions showing the current relationships between program variables.

Zuse's manuscript contained programs of far greater complexity than any written prior to 1945. Included were programs to sort arrays of numbers; test the connectivity of a given graph; carry out integer and floating-point operations, including square root; and perform syntax analysis on logic formulas that had parentheses and operators in six

Downloaded From EnggTree.com

different levels of precedence.

The most remarkable were his 49 pages of algorithms for playing chess, a game in which he was not an expert.

The following example assignment statement, which assigns the value of the expression A[4] +1 to A[5], illustrates this notation. The row labeled V is for subscripts, and the row labeled S is for the data types. In this example, 1.n means an integer of n bits:

	| A + 1 =>	A	
V	| 4	5	(subscript)
S	| 1.n	1.n	(data type)

Pseudocodes

The language discussed in this section is called pseudocodes because that's what they were named at the time they were developed and used (the late 1940s and early 1950s)

There were no high-level programming languages or even assembly languages, so programming was done in machine code, which is both tedious and error prone. The use of numeric codes for specifying instructions. For example, an ADD instruction might be specified by the code 14 rather than a connotative textual name.

A more serious problem is absolute addressing, which makes program modification tedious and error prone. For example, suppose we have a machine language program stored in memory. Many of the instructions in such a program refer to other locations within the program, usually to reference data or to indicate the targets of branch instructions. Inserting an instruction at any position in the program other than at the end invalidates the correctness of all instructions that refer to addresses beyond the insertion point, because those addresses must be increased to make room for the new instruction. To make the addition correctly, all instructions that refer to addresses that follow the addition must be found and modified. A similar problem occurs with deletion of an instruction. In this case, however, machine languages often include a "no operation" instruction that can replace deleted instructions, thereby avoiding the problem. These are standard problems with all machine languages.

Short Code:

The first of these new languages, named Short Code, was developed by John Mauchly in 1949 for the BINAC computer, which was one of the first successful stored-program electronic computers. Short Code was later transferred to a UNIVAC I computer.

The following operation codes were included:

01 -	06 abs value	1n (n+2)nd power
02)	07 +	2n (n+2)nd root
03 =	08 pause	4n if <= n
04 /	09 (	58 print and tab

For example, X0 and Y0 could be variables. The statement

X0 = SQRT(ABS(Y0))

would be coded in a word as 00 X0 03 20 06 Y0. The initial 00 was used as

Downloaded From EnggTree.com

padding to fill the word.

Short Code was not translated to machine code; rather, it was implemented with a pure interpreter. At the time, this process was called automatic programming. It clearly simplified the programming process, but at the expense of execution time. Short Code interpretation was approximately 50 times slower than machine code.

Speed Coding:

The Speedcoding system developed by John Backus for the IBM 701. The Speedcoding interpreter effectively converted the 701 to a virtual three-address floating-point calculator. The system included pseudoinstructions for the four arithmetic operations on floating-point data, as well as operations such as square root, sine, arc tangent, exponent, and logarithm. Conditional and unconditional branches and input/output conversions were also part of the virtual architecture.

Auto-increment registers for array access

Only 700 words left for user program

Related Systems

The UNIVAC Compiling System Developed by a team led by Grace Hopper Pseudocode expanded into machine code

David J. Wheeler (Cambridge University) developed a method of using blocks of re-locatable addresses to solve the problem of absolute addressing

IBM 704 and Fortran

One of the primary reasons why the slowness of interpretive systems was tolerated from the late 1940s to the mid-1950s was the lack of floating-point hardware in the available computers. All floating-point operations had to be simulated in software, a very time-consuming process

Design Process

John Backus and his group at IBM had produced the report titled "The IBM Mathematical FORmula TRANslating System: FORTRAN" . This document described the first version of Fortran, which we refer to as Fortran 0, which is not implemented. The document stated that Fortran would eliminate coding errors and the debugging process. Based on this, the first Fortran compiler included little syntax error checking.

The environment in which Fortran was developed was as follows: (1) Computers had small memories and were slow and relatively unreliable; (2) the primary use of computers was for scientific computations; (3) there were no existing efficient and effective ways to program computers; and (4) because of the high cost of computers compared to the cost of programmers, speed of the generated object code was the primary goal of the first Fortran compilers.

Fortran I Overview

The implementation of Fortran 0 is called the Fortran 1. Variable names of upto six characters (it had been just two in Fortran 0). There were no data-typing statements in

Downloaded From EnggTree.com

the Fortran I language. Variables whose names began with I, J, K, L, M, and N were implicitly integer type, and all others were implicitly floating-point. The machine code produced by the compiler would be about half as efficient as that could be produced by hand. The Fortran development group nearly achieved its goal in efficiency.

Fortran II

The Fortran II compiler was distributed in the spring of 1958. It fixed many of the bugs in the Fortran I compilation system and added some significant features to the language, the most important being the independent compilation of subroutines. Without independent compilation, any change in a program required that the entire program be recompiled.

Fortrans IV, 77, 90, 95, 2003, and 2008

A Fortran III was developed, but it was never widely distributed. Fortran IV, however, became one of the most widely used programming languages of its time. It evolved over the period 1960 to 1962 and was standardized as Fortran 66 (ANSI, 1966), although that name was rarely used. Fortran IV was an improvement over Fortran II in many ways. Among its most important additions were explicit type declarations for variables, a logical If construct, and the capability of passing subprograms as parameters to other subprograms.

Fortran IV was replaced by Fortran 77, which became the new standard in 1978 (ANSI, 1978a). Fortran 77 retained most of the features of Fortran IV and added character string handling, logical loop control statements, and an If with an optional Else clause. Fortran 90 (ANSI, 1992) was dramatically different from Fortran 77. The most significant additions were dynamic arrays, records, pointers, a multiple selection statement, and modules. In addition, Fortran 90 subprograms could be recursively called. A new concept that was included in the Fortran 90 definition was that of removing some language features from earlier versions. While Fortran 90 included all of the features of Fortran 77, the language definition included a list of constructs that were recommended for removal in the next version of the language. Fortran 90 included two simple syntactic changes that altered the appearance of both programs and the literature describing the language. First, the required fixed format of code, which required the use of specific character positions for specific parts of statements, was dropped. For example, statement labels could appear only in the first five positions and statements could not begin before the seventh position. This rigid formatting of code was designed around the use of punch cards. The second change was that the official spelling of FORTRAN became Fortran. This change was accompanied by the change in convention of using all uppercase letters for keywords and identifiers in Fortran programs. The new convention was that only the first letter of keywords and identifiers would be uppercase. Fortran 95 (INCITS/ISO/IEC, 1997) continued the evolution of the language, but only a few changes were made. Among other things, a new iteration construct, Forall, was added to ease the task of parallelizing Fortran programs. Fortran 2003 added support for object-oriented

Downloaded From EnggTree.com

programming, parameterized derived types, procedure pointers, and interoperability with the C programming language.

The latest version of Fortran, Fortran 2008 (ISO/IEC 1539-1, 2010), added support for blocks to define local scopes, co-arrays, which provide a parallel execution model, and the DO CONCURRENT construct, to specify loops without interdependencies.

FUNCTIONAL PROGRAMMING:LISP

The first functional programming language was invented to provide language features for list processing, the need for which grew out of the first applications in the area of artificial intelligence (AI).

The Beginnings of Artificial Intelligence and List Programming

The language, named IPL-I (Information Processing Language I), was never implemented. The next version, IPL-II, was implemented on a RAND Johnniac computer. They were actually assembly languages for a hypothetical computer, implemented with an interpreter, in which list-processing instructions were included.Thus, the Fortran list processing language (FLPL) was designed and implemented as an extension to Fortran. FLPL was used to construct a theorem prover for plane geometry.

Language Overview

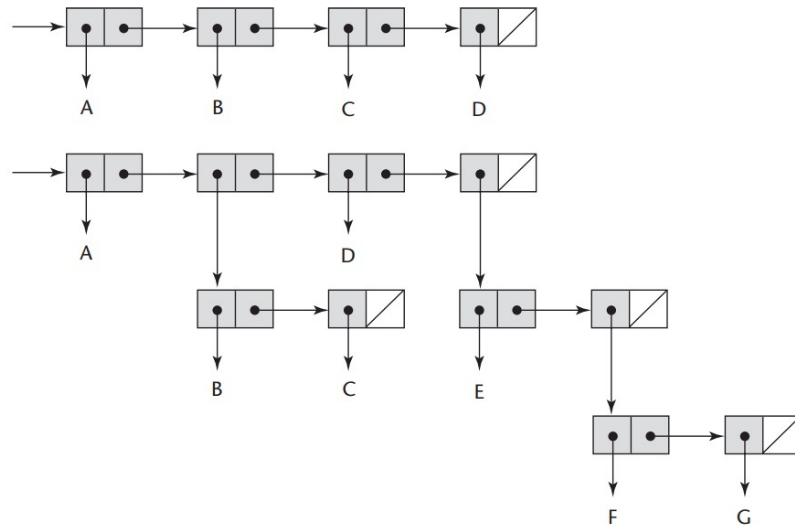
Data Structures

Pure Lisp has only two kinds of data structures: atoms and lists. Atoms are either symbols, which have the form of identifiers, or numeric literals.

Lists are specified by delimiting their elements with parentheses. Simple lists, in which elements are restricted to atoms, have the form (A B C D) Nested list structures are also specified by parentheses. For example, the list (A (B C) D (E (F G)) is composed of four elements. The first is the atom A; the second is the sublist (B C); the third is the atom D; the fourth is the sublist (E (F G)), which has as its second element the sublist (F G). Internally, lists are stored as single-linked list structures, in which each node has two pointers and represents a list element. A node containing an atom has its first pointer pointing to some representation of the atom, such as its symbol or numeric value, or a pointer to a sublist. A node for a sublist element has its first pointer pointing to the first node of the sublist. In both cases, the second pointer of a node points to the next element of the list. A list is referenced by a pointer to its first element.

The internal representations of the two lists shown earlier are depicted in Figure. Note that the elements of a list are shown horizontally. The last element of a list has no successor, so its link is NIL, which is represented in Figure as a diagonal line in the element. Sublists are shown with the same structure

Downloaded From EnggTree.com



Processes in Functional Programming

Lisp was designed as a functional programming language. All computation in a purely functional program is accomplished by applying functions to arguments.Furthermore, repetitive processes can be specified with recursive function calls, making iteration (loops) unnecessary.

The following is an example of a Lisp program:

```
; Lisp Example function
; The following code defines a Lisp predicate function
; that takes two lists as arguments and returns True
; if the two lists are equal, and NIL (false) otherwise
(DEFUN equal_lists (lis1 lis2)
    (COND
        ((ATOM lis1) (EQ lis1 lis2))
        ((ATOM lis2) NIL)
        ((equal_lists (CAR lis1) (CAR lis2))
            (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
    )
)
```

Two Descendants of Lisp

Two dialects of Lisp are now widely used, Scheme and Common Lisp.

Scheme

The Scheme language emerged from MIT in the mid-1970s. It is characterized by its small size, its exclusive use of static scoping and its treatment of functions as first-class entities. As first-class entities, Scheme functions can be assigned to variables, passed as parameters, and returned as the values of function applications.As a small language with simple syntax and semantics, Scheme is well suited to educational applications

Downloaded From EnggTree.com

Common Lisp

During the 1970s and early 1980s, a large number of different dialects of Lisp were developed and used. This led to the familiar problem of lack of portability among programs written in the various dialects. Common Lisp was created in an effort to rectify this situation. Common Lisp was designed by combining the features of several dialects of Lisp developed in the early 1980s, including Scheme, into a single language. Being such an amalgam, Common Lisp is a relatively large and complex language.

ALGOL :

ALGOL was the result of efforts to design a universal programming language for scientific applications.

The following goals for the new language:

• The syntax of the language should be as close as possible to standard mathematical notation, and programs written in it should be readable with little further explanation.

• It should be possible to use the language for the description of algorithms in printed publications.

• Programs in the new language must be mechanically translatable into machine language

ALGOL 58:

- Concept of data type was formalized
- Names could be any length
- Arrays could have any number of subscripts
- Parameters were separated by mode (in & out)
- Subscripts were placed in brackets
- Compound statements (begin ... end)
- Semicolon as a statement separator
- Assignment operator was :=
- if had an else-if clause

ALGOL 60:

The modifications made to ALGOL 58 were dramatic.

Among the most important new developments were the following:

• The concept of block structure was introduced.

• Two different means of passing parameters to subprograms were allowed: pass by value and pass by name.

• Procedures were allowed to be recursive.

• Stack-dynamic arrays were allowed.

It was the first time that an international group attempted to design a programming language. It was the first language that was designed to be machine independent. It was also the first language whose syntax was formally described.

Downloaded From EnggTree.com

There are a number of reasons for its lack of acceptance.

- some of the features of ALGOL 60 turned out to be too flexible
- The lack of input and output statements in the language

COBOL

It was developed in an attempt to design a new language for business applications. One compiled language for business applications is FLOW-MATIC

FLOW-MATIC features

- Names up to 12 characters, with embedded hyphens
- English names for arithmetic operators (no arithmetic expressions)
- The first word in every statement was a verb

COBOL 60:

- First macro facility in a high-level language
- Hierarchical data structures (records)
- Nested selection statements
- Long names (up to 30 characters), with hyphens
- Separate data division

It was the first programming language whose use was mandated by the Department of Defense (DoD).

BASIC:

Basic is another programming language that has enjoyed widespread use like COBOL. It was easy for beginners to learn, especially those who were not science oriented, and its smaller dialects could be implemented on computers with very small memories.

The goals of the system were as follows:

1. It must be easy for non science students to learn and use.
2. It must be "pleasant and friendly."
3. It must provide fast turnaround for homework.
4. It must allow free and private access.
5. It must consider user time more important than computer time.

It is the first widely used language with time sharing

PL/I

PL/I represents the first large-scale attempt to design a language that could be used for a broad spectrum of application areas. All previous and most subsequent languages have focused on one particular application area, such as science, artificial intelligence, or business.

The best single-sentence description of PL/I is that it included what were then considered the best parts of ALGOL 60, Fortran IV, and COBOL 60.

PL/I was the first programming language to have the following facilities:

• Programs were allowed to create concurrently executing subprograms. Although this was a good idea, it was poorly developed in PL/I.

Downloaded From EnggTree.com

• It was possible to detect and handle 23 different types of exceptions, or run-time errors.

• Subprograms were allowed to be used recursively, but the capability could be disabled, allowing more efficient linkage for nonrecursive subprograms.

• Pointers were included as a data type.

• Cross-sections of arrays could be referenced.

Disadvantage with this PL/I is too large and too complex.

APL and SNOBOL:

APL and SNOBOL are quite different. They share two fundamental characteristics, however: dynamic typing and dynamic storage allocation. A variable acquires a type when it is assigned a value. Storage is allocated to a variable only when it is assigned a value, because before that there is no way to know the amount of storage that will be needed.

SIMULA 67:

SIMULA 67 is an extension of ALGOL 60, taking both block structure and the control statements from that language. The primary deficiency of ALGOL 60 for simulation applications was the design of its subprograms. Simulation requires subprograms that are allowed to restart at the position where they previously stopped. Subprograms with this kind of control are known as **coroutines** because the caller and called subprograms have a somewhat equal relationship with each other, rather than the rigid master/slave relationship they have in most imperative languages. To provide support for coroutines in SIMULA 67, the class construct was developed. This was an important development because the concept of data abstraction began with it and data abstraction provides the foundation for object-oriented programming.

C:

C is well suited for a wide variety of applications. C was designed and implemented by Dennis Ritchie at Bell Laboratories in 1972. C has adequate control statements and data-structuring facilities to allow its use in many application areas.

Ada:

The Ada language is the most extensive and expensive language. The Ada language was developed for the Department of Defense (DoD).

Four major contributions of the Ada language.

Packages in the Ada language provide the means for encapsulating data objects, specifications for data types, and procedures. This, in turn, provides the support for the use of data abstraction in program design.

The Ada language includes extensive facilities for exception handling, which allow the programmer to gain control after any one of a wide variety of exceptions, or run-time errors, has been detected.

Program units can be generic in Ada. For example, it is possible to write a sort procedure that uses an unspecified type for the data to be sorted. Such a generic procedure must

Downloaded From EnggTree.com

be instantiated for a specified type before it can be used, which is done with a statement that causes the compiler to generate a version of the procedure with the given type. The availability of such generic units increases the range of program units that might be reused, rather than duplicated, by programmers.

The Ada language also provides for concurrent execution of special program units, named tasks, using the rendezvous mechanism. Rendezvous is the stack-dynamic array is one for which the subscript range or ranges are specified by variables, so that the size of the array is set at the time storage is allocated to the array, which happens when the declaration is reached during execution name of a method of intertask communication and synchronization.

Smalltalk:

Smalltalk was the first programming language that fully supported object-oriented programming. All computing in Smalltalk is done by the same uniform technique: sending a message to an object to invoke one of its methods. A reply to a message is an object, which either returns the requested information or simply notifies the sender that the requested processing has been completed.

The fundamental difference between a message and a subprogram call is this: A message is sent to a data object, specifically to one of the methods defined for the object. The called method is then executed, often modifying the data of the object to which the message was sent; a subprogram call is a message to the code of a subprogram. Usually the data to be processed by the subprogram is sent to it as a parameter.

Smalltalk has done a great deal to promote two separate aspects of computing: graphical user interfaces and object-oriented programming.

C++:

C++ builds language facilities, borrowed from Simula 67, on top of C to support much of what Smalltalk pioneered. C++ has evolved from C through a sequence of modifications to improve its imperative features and to add constructs to support object-oriented programming.

The first step from C toward C++ was made by Bjarne Stroustrup at Bell Laboratories in 1980.

C++ has both functions and methods, it supports both procedural and object-oriented programming.

Operators in C++ can be overloaded, meaning the user can create operators for existing operators on user-defined types. C++ methods can also be overloaded, meaning the user can define more than one method with the same name, provided either the numbers or types of their parameters are different.

Dynamic binding in C++ is provided by virtual methods.

C++ supports multiple inheritance.

Objective C:

Objective-C is another hybrid language with both imperative and object-oriented features. Objective-C was designed by Brad Cox and Tom Love in the early 1980s. Initially, it consist of C plus the classes and message passing of Smalltalk.

Downloaded From EnggTree.com

JAVA:

Java's designers started with C++, removed some constructs, changed some, and added a few others.

Java does not have pointers, but its reference types provide some of the capabilities of pointers.

Java has a primitive Boolean type named boolean, used mainly for the control expressions of its control statements (such as if and while). One significant difference between Java and many of its predecessors that support object-oriented programming, including C++, is that it is not possible to write stand-alone subprograms in Java. Java supports object-oriented programming only.Methods can be called through a class or object only. All Java subprograms are methods and are defined in classes.

Another important difference between C++ and Java is that C++ supports multiple inheritance directly in its class definitions. Java supports only single inheritance of classes, although some of the benefits of multiple inheritance can be gained by using its interface construct.

Java uses implicit storage deallocation for its objects, often called garbage collection. This frees the programmer from needing to delete objects explicitly when they are no longer needed.

Scripting languages:

- **Perl**
  - Designed by Larry Wall—first released in 1987
  - Variables are statically typed but implicitly declared
  - Three distinctive namespaces, denoted by the first character of a variable's name
  - Also used for a replacement for UNIX system administration language
- JavaScript
  - Began at Netscape, but later became a joint venture of Netscape and Sun Microsystems
  - A client-side HTML-embedded scripting language, often used to create dynamic HTML documents
  - Purely interpreted
  - Related to Java only through similar syntax
- PHP
  - PHP: Hypertext Preprocessor, designed by Rasmus Lerdorf
  - A server-side HTML-embedded scripting language, often used for form processing and database access through the Web
- Python
  - An OO interpreted scripting language
  - Type checked but dynamically typed

Downloaded From EnggTree.com

- Used for CGI programming and form processing
- Dynamically typed, but type checked

C#:

C# is based on C++ and Java but includes some ideas from Delphi and Visual Basic. The purpose of C# is to provide a language for component-based software development, specifically for such development in the .NET Framework.

C++ supports multiple inheritance, pointers, structs, enum types, operator overloading, and a goto statement.

C# was meant to be an improvement over both C++ and Java as a general-purpose programming language

Markup-Programming Hybrid Languages:

- XSLT

  eXtensible markup language (XML) is a metamarkup language. Such a language is used to define markup languages. The transformation of XML documents to HTML documents is specified in another markup language, eXtensible stylesheet language transformations (XSLT) (www.w3.org/TR/XSLT). XSLT can specify programming-like operations. Therefore, XSLT is a markup-programming hybrid language.
- JSP
  - Java Server Pages: a collection of technologies to support dynamic Web documents
  - servlet: a Java program that resides on a Web server and is enacted when called by a requested HTML document; a servlet's output is displayed by the browser
  - JSTL includes programming constructs in the form of HTML elements.

DESCRIBING SYNTAX:

The study of programming languages can be divided into examinations of syntax and semantics. The syntax of a programming language is the form of its expressions, statements, and program units. Its semantics is the meaning of those expressions, statements, and program units. For example, the syntax of a Java while statement is

while (boolean_expr) statement

The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed. Then control implicitly returns to the Boolean expression to repeat the process. If the Boolean expression is false, control transfers to the statement following the while construct.

Describing syntax is easier than describing semantics, because a concise and universally accepted notation is available for syntax description, but none has yet been developed for semantics.

Downloaded From EnggTree.com

Terminology:

LEXEME:

The lexeme of a programming language include its numeric literals, operators, and special words, Program is a sequence of lexemes.

TOKEN:

Lexemes are partitioned into groups— for example, the names of variables, methods, classes, and so forth in a programming language form a group called identifiers. Each lexeme group is represented by a name, or token. So, a token of a language is a category of its lexemes. For example, an identifier is a token that can have lexemes such as sum and total.

Consider the following Java statement:

index = 2 * count + 17;

The lexemes and tokens of this statement are

Lexemes	Tokens
index	identifier
=	equal_sign
2	int_literal
*	mult_op
Count	identifier
+	plus_op
17	int_literal
;	semicolon

Language Recognizers:

Languages can be formally defined in two distinct ways: by recognition and by generation. Suppose we have a language L that uses an alphabet Σ of characters. To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet Σ. R would indicate whether a given input string was or was not in L.

The syntax analysis part of a compiler is a recognizer for the language the compiler translates.

Language Generators:

A language generator is a device that can be used to generate the sentences of a language. We can think of the generator as having a button that produces a sentence of the language every time it is pushed. Because the particular sentence that is produced by a generator when its button is pushed is unpredictable.

Methods of Describing Syntax:

The formal language- generation mechanisms, usually called grammars, are commonly used to describe the syntax of programming languages.

Downloaded From EnggTree.com

Backus-Naur Form and Context-Free Grammars:

Context-Free Grammars:

The syntax of whole programming languages, with minor exceptions, can be described by context- free grammars.

Backus-Naur Form:

A paper describing ALGOL 58 was presented by John Backus. This paper introduced a new formal notation for specifying programming language syntax.The new notation was later modified slightly by Peter Naur for the description of ALGOL 60. This revised method of syntax description became known as Backus-Naur Form, or simply BNF. BNF is a natural notation for describing syntax. BNF is nearly identical to Chomsky's generative devices for context- free languages, called context- free grammars.

Fundamentals:

A metalanguage is a language that is used to describe another language. BNF is a metalanguage for programming languages.

BNF uses abstractions for syntactic structures. A simple Java assignment statement, for example,

<assign>→<var> =<expression>

The text on the left side of the arrow, which is called the left-hand side (LHS), is the abstraction being defined. The text to the right of the arrow is the definition of the LHS. It is called the right- hand side (RHS) and consists of some mixture of tokens, lexemes, and references to other abstraction.

The definition is called a rule, or production.

The abstractions in a BNF description, or grammar, are often called nonterminal symbols, or simply nonterminals, and the lexemes and tokens of the rules are called terminal symbols, or simply terminals. A BNF description, or grammar, is a collection of rules.

Multiple definitions can be written as a single rule, with the different definitions separated by | logical OR. For example, a Java if statement can be described with the rules

<if_stmt>→ if (<logic_expr>) <stmt>
<if_stmt>→ if (<logic_expr>) <stmt>else <stmt>

OR

<if_stmt>→ if (<logic_expr>) <stmt>
| if (<logic_expr>) <stmt>else <stmt>

A rule is recursive if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

<ident_list>→ identifier
| identifier, <ident_list>

This defines as either a single token (identifier) or an identifier followed by a comma and another instance of <ident_list>.

Downloaded From EnggTree.com

Grammars and Derivations:

A grammar is a generative device for defining languages. The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the start symbol. This sequence of rule applications is called a derivation.

A Grammar for a Small Language

```
<program> → begin <stmt_list> end

<stmt_list> → <stmt>
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
             | <var> – <var>
             | <var>
```

A derivation of a program in this language follows:

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions. Each of the strings in the derivation, including program, is called a sentential form.

If the Leftmost non terminal is replaced then it is called Left most derivation. The derivation continues until the sentential form contains no nonterminals. That sentential form, consisting of only terminals, or lexemes, is the generated sentence. In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost.

PARSE TREE:

One of the most attractive features of grammars is that they describe the hierarchical syntactic structure of the sentences of the languages called parse trees.

Downloaded From EnggTree.com

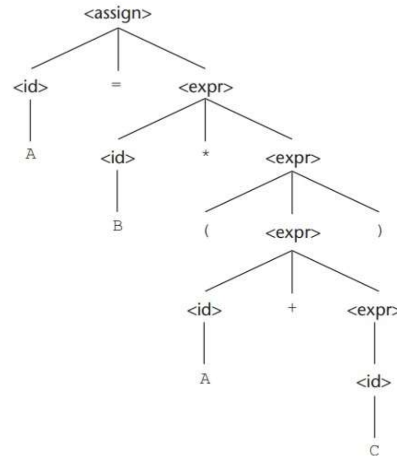


Figure 3.1

A parse tree for the simple statement A = B * (A + C)

Every internal node of a parse tree is labeled with a nonterminal symbol;every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence.

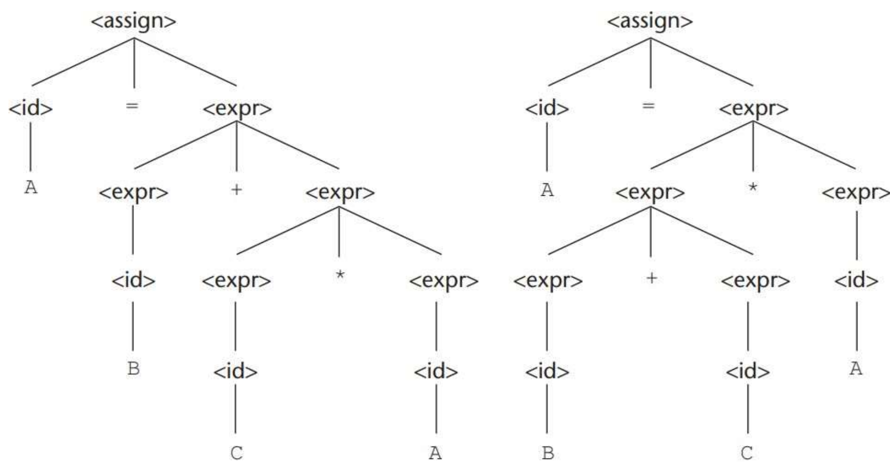
Ambiguity:

A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be ambiguous.

An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

The grammar is ambiguous because the sentence A = B + C * A has two distinct parse trees,



Ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form. Specifically, the compiler chooses the code

Downloaded From EnggTree.com

to be generated for a statement by examining its parse tree. If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.

Several other characteristics of a grammar that are sometimes useful in determining whether a grammar is ambiguous. They include the following: (1) if the grammar generates a sentence with more than one leftmost derivation and (2) if the grammar generates a sentence with more than one rightmost derivation.

Operator Precedence:

When an expression includes two different operators, for example, x + y * z, semantic issue is the order of evaluation of the two operators. This semantic question can be answered by assigning different precedence levels to operators. For example, if * has been assigned higher precedence than +, multiplication will be done first, regardless of the order of appearance of the two operators in the expression.

In the first parse tree , for example, the multiplication operator is generated lower in the tree, which could indicate that it has precedence over the addition operator in the expression. The second parse tree, however, indicates just the opposite.

The correct ordering is specified by using separate nonterminal symbols to represent the operands of the operators that have different precedence. This requires additional nonterminals and some new rules.

An Unambiguous Grammar for Expressions

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
        | <term>
<term> → <term> * <factor>
        | <factor>
<factor> → ( <expr>)
          | <id>
```

This grammar generates the same language as the grammars of previous example, but it is unambiguous and it specifies the usual precedence order of multiplication and addition operators. The following derivation of the sentence A = B + C * A

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <term>
         => A = <term> + <term>
         => A = <factor> + <term>
         => A = <id> + <term>
         => A = B + <term>
         => A = B + <term> * <factor>
         => A = B + <factor> * <factor>
         => A = B + <id> * <factor>
         => A = B + C * <factor>
         => A = B + C * <id>
         => A = B + C * A
```

Downloaded From EnggTree.com

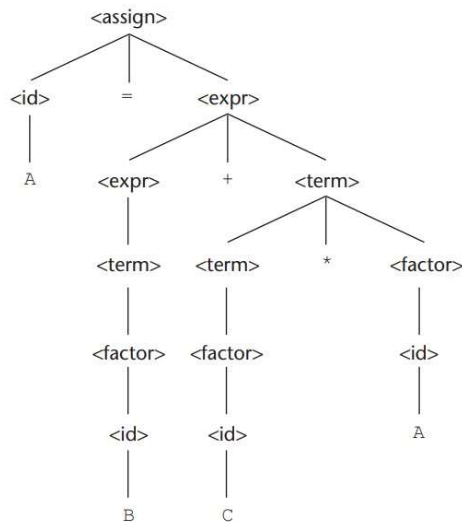
The connection between parse trees and derivations is very close. Either can easily be constructed from the other. Every derivation with an unambiguous grammar has a unique parse tree.

```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C * A
         => <id> = <term> + C * A
         => <id> = <factor> + C * A
         => <id> = <id> + C * A
         => <id> = B + C * A
         => A = B + C * A
```

Both the derivations are represented by the same parse tree.

Figure 3.3

The unique parse tree for A = B + C * A using an unambiguous grammar



Associativity of Operators:

When an expression includes two operators that have the same precedence.

For example,

A / B * C

a semantic rule is required to specify which should have precedence. This rule is named associativity.

Downloaded From EnggTree.com

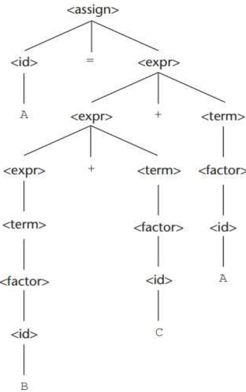


Figure 3.4 A parse tree for A = B + C + A illustrating the associativity of addition

The parse tree of Figure 3.4 shows the left addition operator lower than the right addition operator. This is the correct order if addition is meant to be left associative. In mathematics, addition is associative, which means that left and right associative orders of evaluation mean the same thing. That is,

(A + B) + C = A + (B + C).

When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be left recursive. This left recursion specifies left associativity. When one of these algorithms is to be used, the grammar must be modified to remove the left recursion.

A grammar rule is right recursive if the LHS appears at the right end of the RHS. Rules such as <factor>→<exp> ** <factor>

| <exp>

<exp>→(<exp>)

| id

could be used to describe exponentiation as a right-associative operator.

An Unambiguous Grammar for if-else

The BNF rules for a Java if-else statement are as follows:

<if_stmt>→ if (<logic_expr>)<stmt>

if (<logic_expr>)<stmt> else <stmt>

If we also have <stmt>→ <if_stmt>, this grammar is ambiguous.

The simplest sentential form that illustrates this ambiguity is

if (<logic_expr>)if (<logic_expr>)<stmt> else<stmt>

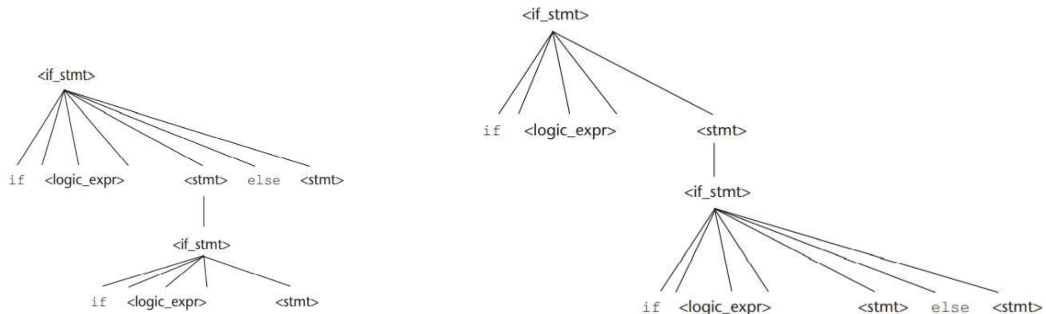


Figure 3.5 Two distinct parse trees for the same sentential form

The unambiguous grammar is as follows:

Downloaded From EnggTree.com

EnggTree.com

<stmt> → <matched> | <unmatched>
<matched> → **if** (<logic_expr>) <matched> **else** <matched>
| any non-if statement
<unmatched> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <matched> **else** <unmatched>

Extended BNF:

Because of a few minor inconveniences in BNF, it has been extended called Extended BNF, or simply EBNF. The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability.

Three extensions are commonly included in the various versions of EBNF. The first of these denotes an optional part of an RHS, which is delimited by brackets. For example, a C if-else statement can be described as

<if_stmt>→ if (<expression>) <statement>[else <statement>]

Without the use of the brackets, the syntactic description of this statement would require the following two rules:

<if_stmt>→ if (<expression>) <statement>
| if (<expression>) <statement> else <statement>

The second extension is the use of braces in an RHS to indicate that the enclosed part can be repeated indefinitely. For example, lists of identifiers separated by commas can be described by the following rule: <ident_list>→<identifier>{<identifier>}

The part enclosed within braces can be iterated any number of times.

The third common extension deals with multiple-choice options. When a single element must be chosen from a group, the options are placed in parentheses and separated by the OR operator, |.

For example, <term>→<term>(*|/|%) <factor>

In BNF, a description of this would require the following three rules:

<term>→<term> *<factor>
|<term>/<factor>
|<term>%<factor>

BNF and EBNF Versions of an Expression Grammar

BNF:
<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → <exp> ** <factor>
<exp>
<exp> → (<expr>)
| id

EBNF:
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

Downloaded From EnggTree.com

The BNF rule <expr>→<expr> + <term> clearly specifies the + operator to be left associative. However, the EBNF version, <expr>→<term> {+ <term>} does not imply the direction of associativity.

Some versions of EBNF allow a numeric superscript to be attached to the right brace to indicate an upper limit to the number of times the enclosed part can be repeated. Also, some versions use a plus (+) superscript to indicate one or more repetitions. For example,
<compound>→ begin <stmt> {<stmt>} end
and
<compound>→ begin $\{$<stmt>$\}^+$ end
some variations on BNF and EBNF have appeared.
These are the following:
• In place of the arrow, a colon is used and the RHS is placed on the next line.
• Instead of a vertical bar to separate alternative RHSs, they are simply placed on separate lines.
• In place of square brackets to indicate something being optional, the subscript opt is used.
For example, Constructor Declarator → SimpleName ($FormalParameterList_{opt}$)
• Rather than using the | symbol in a parenthesized list of elements to indicate a choice, the words "one of" are used.
For example, AssignmentOperator → one of = *= /= %= += -= <<= >>= &= ^= |=

ATTRIBUTE GRAMMARS

An attribute grammar is a device used to describe more of the structure of a programming language that can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar.

Static Semantics:

There are some characteristics of programming languages that are difficult to describe with BNF, and some that are impossible.

In Java, for example, a floating- point value cannot be assigned to an integer type variable, although the opposite is legal. It requires additional nonterminal symbols and rules to specify in BNF.

Consider the common rule that all variables must be declared before they are referenced. It has been proven that this rule cannot be specified in BNF.

The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics).

Because of the problems of describing static semantics with BNF, a variety of more powerful mechanisms has been devised for that task. One such mechanism, attribute grammars, was designed by Knuth (1968a) to describe both the syntax and the static semantics of programs.

Downloaded From EnggTree.com

Attribute grammars are a formal approach both to describe and check the correctness of the static semantics rules of a program.

BASIC CONCEPTS:

Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate functions. Attributes, which are associated with grammar symbols (the terminal and nonterminal symbols), are similar to variables that have values assigned to them. Attribute computation functions, sometimes called semantic functions, are associated with grammar rules. They are used to specify how attribute values are computed. Predicate functions, which state the static semantic rules of the language, are associated with grammar rules.

Attribute Grammars Defined:

An attribute grammar is a grammar with the following additional features:

• Associated with each grammar symbol X is a set of attributes A(X). The set A(X) consists of two disjoint sets S(X) and I(X), called synthesized and inherited attributes, respectively. Synthesized attributes are used to pass semantic information up a parse tree, while inherited attributes pass semantic information down and across a tree.

• Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule. For a rule $X_0 \rightarrow X_1,\ldots,X_n$, the synthesized attributes of X_0 are computed with semantic functions of the form

$S(X_0) = f(A(X_1), \ldots, A(X_n))$.

So the value of a synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes.

Inherited attributes of symbols Xj , $1 \leq j \leq n$ (in the rule above), are computed with a semantic function of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$. So the value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes.

Note that, to avoid circularity, inherited attributes are often restricted to functions of the form

$I(X_j) = f(A(X_0), \ldots, A(X_{(j-1)}))$. This form prevents an inherited attribute from depending on itself or on attributes to the right in the parse tree.

A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node. If all the attribute values in a parse tree have been computed, the tree is said to be fully attributed.

Intrinsic Attributes:

Intrinsic attributes are synthesized attributes of leaf nodes whose values are determined outside the parse tree.

Example:

The syntax portion of our example attribute grammar is

<assign>→ <var> = <expr>

<expr>→ <var>+<var>

Downloaded From EnggTree.com

|<var>

<var> → *A* | *B* | *C*

The attributes for the nonterminals in the example attribute grammar are described in the following paragraphs:

• actual_type—A synthesized attribute associated with the nonterminals <var>and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the <expr>nonterminal.

• expected_type—An inherited attribute associated with the nonterminal<expr> . It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type.

The process of computing the attribute values of a parse tree, which is sometimes called decorating the parse tree. The following is an evaluation of the attributes, in an order in which it is possible to compute them:

1. <var>.actual_type ← look-up(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(A) (Rule 4)
   <var>[3].actual_type ← look-up(B) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either
   TRUE or FALSE (Rule 2)

Downloaded From EnggTree.com

Figure 3.6

A parse tree for
A = A + B

Figure 3.7

The flow of attributes in the tree

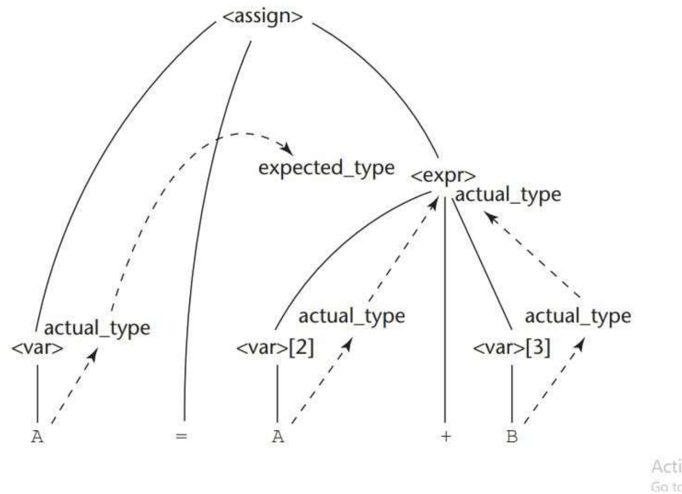
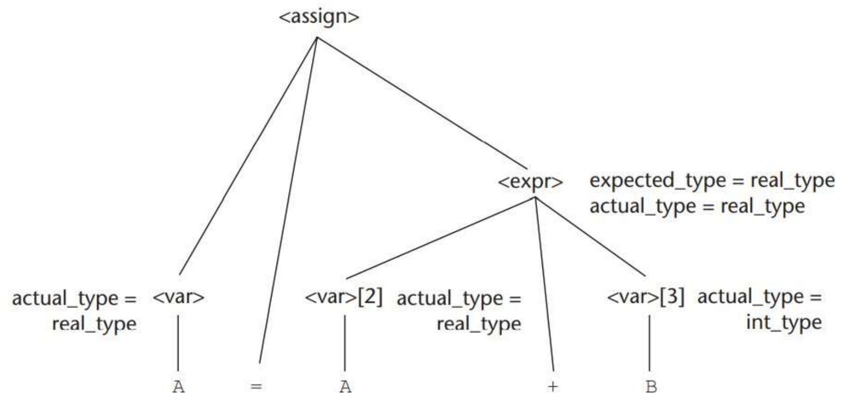


Figure 3.8

A fully attributed parse tree



DESCRIBING SEMANTICS:

Dynamic Semantics is the meaning of the expressions, statements, and program units. There is no universally accepted notation or approach has been devised for dynamic semantics.

Need for the notation/methodology for describing semantics:

- Programmers need to know what statements mean
- Compiler writers must know exactly what language constructs do
- Correctness proofs would be possible

Downloaded From EnggTree.com

EnggTree.com

- Compiler generators would be possible
- Designers could detect ambiguities and inconsistencies

Operational semantics:

Operational semantics is to describe the meaning of a statement or program by specifying the effects of running it on a machine. The effects on the machine are viewed as the sequence of changes in its state, where the machine's state is the collection of the values in its storage.

There are different levels of uses of operational semantics. At the highest level, the interest is in the final result of the execution of a complete program. This is sometimes called natural operational semantics. At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed. This use is sometimes called structural operational semantics.

The first step in creating an operational semantics description of a language is to design an appropriate intermediate language. If the semantics description is to be used for natural operational semantics, a virtual machine (an interpreter) must be constructed for the intermediate language. The virtual machine can be used to execute either single statements, code segments, or whole programs.

For example, the semantics of the C for construct can be described in terms of simpler statements, as in

```
C Statement                          Meaning
for (expr1; expr2; expr3) {                 expr1;
  . . .                              loop:  if expr2 == 0 goto out
}                                           . . .
                                            expr3;
                                            goto loop
                                     out:   . . .
```

The intermediate language and its associated virtual machine used for formal operational semantics descriptions are often highly abstract.

consider the following list of statements, which would be adequate for describing the semantics of the simple control statements of a typical programming language:

ident = var

ident = ident + 1

ident = ident – 1

goto label

if var relop var goto label

In these statements, relop is one of the relational operators from the set {=, <>, >, =, <=}, ident is an identifier, and var is either an identifier or a constant. These statements are all simple and therefore easy to understand and implement. A slight generalization of these three assignment statements allows more general arithmetic expressions and assignment statements to be described. The new statements are

ident = var bin_op var

ident = un_op var

Downloaded From EnggTree.com

where bin_op is a binary arithmetic operator and un_op is a unary operator.

Denotational Semantics:

Denotational semantics is the most rigorous and most widely known formal method for describing the meaning of programs. It is solidly based on recursive function theory. The process of constructing a denotational semantics specification for a programming language requires one to define for each language entity both a mathematical object and a function that maps instances of that language entity onto instances of the mathematical object. The difficulty with this method lies in creating the objects and the mapping functions.

The mapping functions of a denotational semantics programming language specification have a domain and a range. The domain is the collection of values that are legitimate parameters to the function; the range is the collection of objects to which the parameters are mapped. In denotational semantics, the domain is called the syntactic domain, because it is syntactic structures that are mapped. The range is called the semantic domain.

The State of a Program:

Denotational semantics uses the state of the program to describe meaning, whereas operational semantics uses the state of a machine. The key difference between operational semantics and denotational semantics is that state changes in operational semantics are defined by coded algorithms, written in some programming language, whereas in denotational semantics, state changes are defined by mathematical functions. Let the state s of a program be represented as a set of ordered pairs, as follows:

$s = \{<i_1,v_1> <i_2, v_2>. . . <i_n,v_n>\}$

Each i is the name of a variable, and the associated v's are the current values of those variables.

Let VARMAP be a function of two parameters: a variable name and the program state.

The value of VARMAP (i_j , s) is v_j (the value paired with ij in state s).

Most semantics mapping functions for programs and program constructs map states to states. These state changes are used to define the meanings of programs and program constructs. Some language constructs—for example, expressions—are mapped to values, not states.

Expressions:

Expressions are fundamental to most programming languages. The only operators are + and *, and an expression can have at most one operator; the only operands are scalar integer variables and integer literals; there are no parentheses; and the value of an expression is an integer. Following is the BNF description of these expressions:

<expr>→ <dec_num>|<var>|<binary_expr>

<binary_expr>→<left_expr><operator><right_expr>

<right-expr>→<dec_num>|<var>

<right_expr>→<dec_num>|<var>

<operator>→+|*

Downloaded From EnggTree.com

The only error we consider in expressions is a variable having an undefined value. Let Z be the set of integers, and let error be the error value. Then Z ^{U} {error} is the semantic domain for the denotational specification for our expressions.

The mapping function for a given expression E and state s follows. To distinguish between mathematical function definitions and the assignment statements of programming languages, we use the symbol Δ = to define mathematical functions. The implication symbol, =>, used in this definition connects the form of an operand with its associated case (or switch) construct. Dot notation is used to refer to the child nodes of a node.

For example, <binary_expr>.<left_expr> refers to the left child node of <binary_expr>

```
Me (<expr>, s)  Δ= case <expr> of
                     <dec_num>=>Mdec (<dec_num>, s)
                     <var> =>if VARMAP (<var>, s)  == undef
                              then error
                              else VARMAP (<var>, s)
                     <binary_expr> =>
                      if(Me (<binary_expr>.<left_expr>,s)  == undef OR
                         Me (<binary_expr>.<right_expr>, s)  == undef)
                      then error
                      else if  (<binary_expr>.<operator> == '+')
                            then Me (<binary_expr>.<left_expr>, s)  +
                                 Me (<binary_expr>.<right_expr>, s)
                            else Me (<binary_expr>.<left_expr>, s)  *
                                 Me (<binary_expr>.<right_expr>, s)
```

Assignment Statements:

An assignment statement is an expression evaluation plus the setting of the target variable to the expression's value.

```
Ma (x = E, s)  Δ= if Me (E, s)  == error
                 then error
                 else s' = {<i1, v1' >, <i2, v2'>, . . . , <in, vn'>}, where
                       for j = 1, 2, . . . , n
                        if ij == x
                          then vj' = Me (E, s)
                          else vj' = VARMAP (ij, s)
```

Logical Pretest Loops:

The denotational semantics of a logical pretest loop is deceptively simple. There are two other existing mapping functions, Msl and Mb, that map statement lists and states to states and Boolean expressions to Boolean values (or error), respectively.

```
Ml (while B do L, s)  Δ= if Mb (B, s)  == undef
                         then error
                         else if Mb (B, s)  == false
                               then s
                               else if Msl (L, s)  == error
                                     then error
                                     else Ml (while B do L, Msl (L, s) )
```

The meaning of the loop is simply the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors.In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping

Downloaded From EnggTree.com

functions.Recursion, when compared to iteration, is easier to describe with mathematical rigor.

Evaluation:

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, it is of little use to language users.

Axiomatic semantics:

Axiomatic semantics specifies what can be proven about the program. The use of semantic specifications is to prove the correctness of programs.

In axiomatic semantics, there is no model of the state of a machine or program or model of state changes that take place when the program is executed. The meaning of a program is based on relationships among program variables and constants, which are the same for every execution of the program.

Axiomatic semantics has two distinct applications: program verification and program semantics specification. This section focuses on program verification in its description of axiomatic semantics.

The logical expressions used in axiomatic semantics are called predicates, or assertions. An assertion immediately following a statement describes the new constraints on those variables (and possibly others) after execution of the statement. These assertions are called the precondition and postcondition, respectively, of the statement. For two adjacent statements, the postcondition of the first serves as the precondition of the second.
consider the following assignment statement and postcondition:
sum = 2 * x + 1 {sum > 1}
Precondition and postcondition assertions are presented in braces to distinguish them from parts of program statements. One possible precondition for this statement is {x > 10}.

The weakest precondition is the least restrictive precondition that will guarantee the validity of the associated postcondition. For example, in the statement and postcondition given {x > 10}, {x > 50}, and {x > 1000} are all valid preconditions. The weakest of all preconditions in this case is {x > 0}.

An inference rule is a method of inferring the truth of one assertion on the basis of the values of other assertions. The general form of an inference rule is as follows:

$$\frac{S1, S2, \ldots, Sn}{S}$$

This rule states that if S1, S2, . . . , and Sn are true, then the truth of S can be inferred. The top part of an inference rule is called its antecedent; the bottom part is called its consequent. An axiom is a logical statement that is assumed to be true. Therefore, an axiom is an inference rule without an antecedent.

Downloaded From EnggTree.com

EnggTree.com

Assignment Statements:

The precondition and postcondition of an assignment statement together define its meaning. To define the meaning of an assignment statement there must be a way to compute its precondition from its postcondition. Let x = E be a general assignment statement and Q be its postcondition. Then, its weakest precondition, P, is defined by the axiom

$$P = Q_{x \to E}$$

which means that P is computed as Q with all instances of x replaced by E. For example, if we have the assignment statement and postcondition

a = b / 2 - 1 {a < 10}

the weakest precondition is computed by substituting b / 2 - 1 for a in the postcondition {a < 10}, as follows:

b / 2 - 1 < 10

b < 22

Thus, the weakest precondition for the given assignment statement and postcondition is {b < 22}.

The usual notation for specifying the axiomatic semantics of a given statement form is {P} S {Q} where P is the precondition, Q is the postcondition, and S is the statement form. In the case of the assignment statement, the notation is $\{Q_{x \to E}\}$ x = E{Q}

Next, consider the following logical statement:

{x > 5} x = x - 3 {x > 0}

In this case, the given precondition, {x > 5}, is not the same as the assertion produced by the axiom. However, it is obvious that {x > 5} implies {x > 3}. To use this in a proof, an inference rule named the rule of consequence is needed. The form of the rule of consequence is

$$\frac{\{P\}S\{Q\}, P' => P, Q => Q'}{\{P'\}S\{Q'\}}$$

The => symbol means "implies," and S can be any program statement. The rule can be stated as follows: If the logical statement {P}S{Q} is true, the assertion P′ implies the assertion P, and the assertion Q implies the assertion Q′, then it can be inferred that {P′}S{Q′}.

Sequences:

The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence. In this case, the precondition can only be described with an inference rule. Let S1 and S2 be adjacent program statements. If S1 and S2 have the following pre- and postconditions

{P1} S1 {P2}

{P2} S2 {P3}

the inference rule for such a two-statement sequence is

Downloaded From EnggTree.com

$$\frac{\{P1\}S1\{P2\},\{P2\}S2\{P3\}}{\{P1\},S1,S2\{P3\}}$$

So, for our example, {P1} S1; S2 {P3} describes the axiomatic semantics of the sequence S1; S2.
If S1 and S2 are the assignment statements
x1 = E1 and
x2 = E2
then we have
$\{P3_{x2\rightarrow E2}\}$ x2 = E2 {P3}
$\{(P3_{x2\rightarrow SE2})_{x1\rightarrow E1}\}$ x1 = E1 $\{P3_{x2S\rightarrow E2}\}$
Therefore, the weakest precondition for the sequence x1 = E1; x2 = E2 with postcondition P3 is $\{(P3_{x2\rightarrow SE2})_{x1\rightarrow E1}\}$

Selection:

The inference rule for selection statements, the general form of which is if B then S1 else S2.
We consider only selections that include else clauses. The inference rule is

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

This rule specifies that selection statements must be proven both when the Boolean control expression is true and when it is false. The first logical statement above the line represents the then clause; the second represents the else clause.
Consider the following example of the computation of the precondition using the selection inference rule. The example selection statement is
if x > 0
then
y = y - 1
else
y = y + 1
Suppose the postcondition, Q, for this selection statement is {y > 0}. We can use the axiom for assignment on the then clause
y = y - 1 {y > 0}
This produces
{y - 1 > 0} or {y > 1}.
It can be used as the P part of the precondition for the then clause. Now we apply the same axiom to the else clause

y = y + 1 {y > 0}
which produces the precondition {y + 1 > 0} or {y > -1}. Because {y > 1} => {y > -1}, the rule of consequence allows us to use {y > 1} for the precondition of the whole selection statement.

Downloaded From EnggTree.com

Logical Pretest Loops:
Another essential construct of imperative programming languages is the logical pretest, or while loop. The corresponding step in the axiomatic semantics of a while loop is finding an assertion called a loop invariant, which is crucial to finding the weakest precondition. The inference rule for computing the precondition for a while loop is as follows:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{I \text{ and } (\text{not } B)\}}$$

In this rule, I is the loop invariant. This seems simple, but it is not. The complexity lies in finding an appropriate loop invariant. The axiomatic description of a while loop is written as
{P} while B do S end {Q}
The complicating factor for while loops is the question of loop termination. A loop that does not terminate cannot be correct, and in fact computes nothing. If Q is the postcondition that holds immediately after loop exit, then a precondition P for the loop is one that guarantees Q at loop exit and also guarantees that the loop terminates.
The complete axiomatic description of a while construct requires all of the following to be true, in which I is the loop invariant:
P => I
{I and B} S {I}
(I and (not B)) => Q
the loop terminate
If a loop computes a sequence of numeric values, it may be possible to find a loop invariant using an approach that is used for determining the inductive hypothesis when mathematical induction is used to prove a statement about a mathematical sequence. The relationship between the number of iterations and the precondition for the loop body is computed for a few cases, with the hope that a pattern emerges that will apply to the general case. It is helpful to treat the process of producing a weakest precondition as a function, wp. In general
wp(statement, postcondition) = precondition

A wp function is often called a predicate transformer, because it takes a predicate, or assertion, as a parameter and returns another predicate.

LEXICAL ANALYSIS:

A lexical analyzer is a pattern matcher. A pattern matcher attempts to find a substring of a given string of characters that matches a given character pattern.

A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis. A lexical analyzer performs syntax analysis at the lowest level of program structure. An input program appears to a compiler as a single string of characters. The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure. These logical groupings are named lexemes, and the internal codes for categories of these groupings are named tokens. Lexemes are

Downloaded From EnggTree.com

EnggTree.com

recognized by matching the input character string against character string patterns. Although tokens are usually represented as integer values, for the sake of readability of lexical and syntax analyzers, they are often referenced through named constants. Consider the following example of an assignment statement:
result = oldsum - value / 100;
Following are the tokens and lexemes of this statement:

Token	Lexeme
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program. Also, the lexical analyzer inserts lexemes for user- defined names into the symbol table, which is used by later phases of the compiler. Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user.

There are three approaches to building a lexical analyzer:
1. Write a formal description of the token patterns of the language using a descriptive language related to regular expressions. These descriptions are used as input to a software tool that automatically generates a lexical analyzer. There are many such tools available for this. The oldest of these, named lex, is commonly included as part of UNIX systems.
2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

A state transition diagram, or just state diagram, is a directed graph. The nodes of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause the transitions among the states. An arc may also include actions the lexical analyzer must perform when the transition is taken. State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognize members of a class of languages called regular languages. Regular grammars are generative devices for regular languages. The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

Downloaded From EnggTree.com

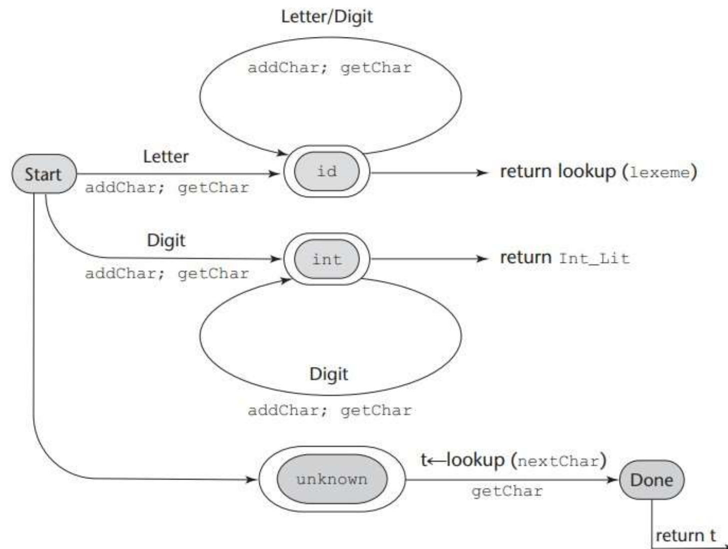


Figure 4.1
A state diagram to recognize names, parentheses, and arithmetic operators

A lexical analyzer often is responsible for the initial construction of the symbol table, which acts as a database of names for the compiler. The entries in the symbol table store information about user- defined names, as well as the attributes of the names. For example, if the name is that of a variable, the variable's type is one of its attributes that will be stored in the symbol table. Names are usually placed in the symbol table by the lexical analyzer. The attributes of a name are usually put in the symbol table by some part of the compiler that is subsequent to the actions of the lexical analyzer.

PARSING:

The process of analyzing syntax that is referred to as syntax analysis is often called parsing. Parsers for programming languages construct parse trees for given program. Parse tree is only implicitly constructed. The information required to build the parse tree is created during the parse. Both parse trees and derivations include all of the syntactic information needed by a language processor.

There are two distinct goals of syntax analysis:

First, the syntax analyzer must check the input program to determine whether it is syntactically correct. When an error is found, the analyzer must produce a diagnostic message and recover.

The second goal of syntax analysis is to produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input. The parse tree (or its trace) is used as the basis for translation.

Parsers are categorized according to the direction in which they build parse trees. The two broad classes of parsers are top-down, in which the tree is built from the root downward to the leaves, and bottom-up, in which the parse tree is built from the leaves upward to the root.

Notational conventions for grammar symbols and strings:

1. Terminal symbols—lowercase letters at the beginning of the alphabet (a, b, . . .)
2. Nonterminal symbols—uppercase letters at the beginning of the alphabet (A, B, . . .)
3. Terminals or nonterminals—uppercase letters at the end of the alphabet (W, X, Y, Z)

Downloaded From EnggTree.com

4. Strings of terminals—lowercase letters at the end of the alphabet (w, x, y, z)
5. Mixed strings (terminals and/or nonterminals)—lowercase Greek letters (α,β,γ,δ)
The nonterminal symbols of programming languages are usually connotative names or abbreviations, surrounded by angle brackets—for example, , , and . The sentences of a language (programs, in the case of a programming language) are strings of terminals. Mixed strings describe right- hand sides (RHSs) of grammar rules and are used in parsing algorithms.

Top-Down Parser:

A top-down parser traces or builds a parse tree in preorder. A preorder traversal of a parse tree begins with the root. Each node is visited before its branches are followed. Branches from a particular node are followed in left-to-right order. This corresponds to a leftmost derivation.
Different top-down parsing algorithms use different information to make parsing decisions. The most common top- down parsers choose the correct RHS for the leftmost nonterminal in the current sentential form by comparing the next token of input with the first symbols that can be generated by the RHSs of those rules.

Bottom-Up Parsers:

A bottom- up parser constructs a parse tree by beginning at the leaves and progressing toward the root. This parse order corresponds to the reverse of a rightmost derivation.
Consider the following grammar and derivation:
S→ aAc
A → aA | b
S => aAc => aaAc => aabc
A bottom-up parser of this sentence, aabc, starts with the sentence and must find the handle in it. In this example, this is an easy task, for the string contains only one RHS, b. When the parser replaces b with its LHS, A, it gets the second to last sentential form in the derivation, aaAc. In the general case, as stated previously, finding the handle is much more difficult, because a sentential form may include several different RHSs.
The most common bottom- up parsing algorithms are in the LR family, where the L specifies a left-to-right scan of the input and the R specifies that a rightmost derivation is generated.

The Recursive-Descent Parser:

A recursive- descent parser is so named because it consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order. This recursion is a reflection of the nature of programming languages, which include several different kinds of nested structures. The syntax of these structures is naturally described with recursive grammar rules.
EBNF is ideally suited for recursive-descent parsers. Consider the following examples:
<if_statement>→ if<logic_expr> <statement> [else <statement]
<ident_list> → ident {, ident}

Downloaded From EnggTree.com

In the first rule, the else clause of an if statement is optional. In the second, an is an identifier, followed by zero or more repetitions of a comma and an identifier.

A recursive-descent parser has a subprogram for each nonterminal in its associated grammar. The responsibility of the subprogram associated with a particular nonterminal is as follows: When given an input string, it traces out the parse tree that can be rooted at that nonterminal and whose leaves match the input string.

Consider the following EBNF description of simple arithmetic expressions:

<expr>→ <term>{(+ | -) <term>}

<term>→ <factor>{(* | /)<factor> }

<factor>→ id | int_constant | (<expr>)

In the following recursive-descent function, expr, the lexical analyzer is the function It gets the next lexeme and puts its token code in the global variable nextToken.

For each terminal symbol in the RHS, that terminal symbol is compared with nextToken. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get the next input token. For each nonterminal, the parsing subprogram for that nonterminal is called.

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
   */
void  expr() {
  printf("Enter <expr>\n");

/* Parse the first term */
  term();

/* As long as the next token is + or -, get
   the next token and parse the next term */
  while  (nextToken == ADD_OP || nextToken == SUB_OP) {
    lex();
    term();
  }
  printf("Exit <expr>\n");
}  /* End of function expr */


/* term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>)
   */
void  term() {
  printf("Enter <term>\n");

/* Parse the first factor */
  factor();

/* As long as the next token is * or /, get the
   next token and parse the next factor */
  while  (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
}  /* End of function term */
```

Downloaded From EnggTree.com

```
/* factor
   Parses strings in the language generated by the rule:
   <factor> -> id | int_constant | ( <expr )
   */
void factor() {
  printf("Enter <factor>\n");

/* Determine which RHS */
  if (nextToken == IDENT || nextToken == INT_LIT)

/* Get the next token */
    lex();

/* If the RHS is ( <expr>), call lex to pass over the
   left parenthesis, call expr, and check for the right
   parenthesis */
  else {
    if (nextToken == LEFT_PAREN) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN)
        lex();
      else
        error();
    } /* End of if (nextToken == ... */

/* It was not an id, an integer literal, or a left
   parenthesis */
    else
      error();
  } /* End of else */

  printf("Exit <factor>\n");;
} /* End of function factor */
```

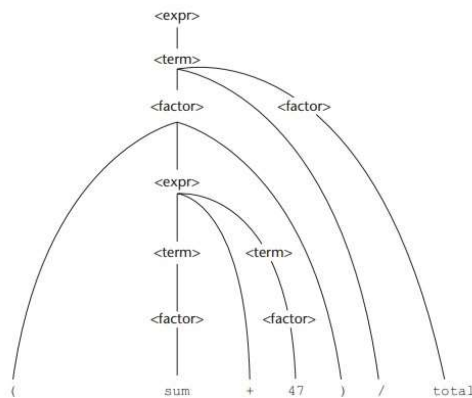


Figure 4.2
Parse tree for (sum + 47) / total

LL Grammar:

One simple grammar characteristic that causes a problem for LL parsers is left recursion. For example, consider the following rule:

A → A + B

A recursive-descent parser subprogram for A immediately calls itself to parse the first symbol in its RHS. That activation of the A parser subprogram then immediately calls itself again, and again, and so forth. It is easy to see that this leads nowhere (except to stack overflow). The left recursion in the rule A → A + B is called direct left recursion, because it occurs in one rule. Direct left recursion can be

Downloaded From EnggTree.com

eliminated from a grammar by the following process:
For each nonterminal, A,
1. Group the A-rules as $A \rightarrow A\alpha 1, | \ldots\ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$ where none of the β's begins with A
2. Replace the original A-rules with
$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots . | \beta_n A'$
$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \alpha_m A' | \varepsilon$
ε specifies the empty string. A rule that has ε as its RHS is called an erasure rule.

Consider the following example grammar and the application of the above process:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

For the E-rules, we have $\alpha_1 = + T$ and $\beta = T$, so we replace the E-rules with

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$

For the T-rules, we have $\alpha_1 = *F$ and $\beta = F$, so we replace the T-rules with

$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$

Because there is no left recursion in the F-rules, they remain the same, so the complete replacement grammar is

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Left recursion is not the only grammar trait that disallows top-down parsing. Another is whether the parser can always choose the correct RHS on the basis of the next token of input, using only the first token generated by the leftmost nonterminal in the current sentential form. There is a relatively simple test of a non-left recursive grammar that indicates whether this can be done, called the pairwise disjointness test. This test requires the ability to compute a set based on the RHSs of a given nonterminal symbol in a grammar. These sets, which are called FIRST, are defined as

$$FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\} \text{ (If } \alpha \Rightarrow^* \varepsilon, \varepsilon \text{ is in } FIRST(\alpha))$$

The pairwise disjointness test is as follows: For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$FIRST(\alpha_i) \cup FIRST(\alpha_j) = \phi$$

Consider the following rules: $A \rightarrow aB \mid bAb \mid Bb$

$B \rightarrow cB \mid d$

The FIRST sets for the RHSs of the A- rules are {a}, {b}, and {c}, {d}, which are clearly disjoint. Therefore, these rules pass the pairwise disjointness test.

Consider the following rules: $A \rightarrow aB \mid BAb$

$B \rightarrow aB \mid b$

Downloaded From EnggTree.com

EnggTree.com

The FIRST sets for the RHSs in the A- rules are {a} and {a}, {b} which are clearly not disjoint. So, these rules fail the pairwise disjointness test.
A grammar that fails the pairwise disjointness test can be modified so that it will pass the test. For example, consider the rule

<variable>→ identifier | identifier [<expression>]

This states that a <variable>is either an identifier or an identifier followed by an expression in brackets (a subscript). These rules clearly do not pass the pairwise disjointness test, because both RHSs begin with the same terminal, identifier. This problem can be alleviated through a process called left factoring.
The two rules can be replaced by the following two rules:

<variable>→ identifier <new>
<new>→ε | [<expression>]

these two pass the pairwise disjointness test.

Bottom-Up Parsing:

Consider the following grammar for arithmetic expressions:

E →E + T | T
T → T * F | F
F → (E) | id

This grammar is left recursive, which is acceptable to bottom- up parsers.
The following rightmost derivation illustrates this grammar:

E => E + T
=> E + T * F
=> E + T * id
=> E + F * id
=> E + id * id
=> T + id * id
=> F + id * id
=> id + id * id

The underlined part of each sentential form in this derivation is the RHS that is rewritten as its corresponding LHS to get the previous sentential form. The process of bottom-up parsing produces the reverse of a rightmost derivation. So, in the example derivation, a bottom-up parser starts with the last sentential form (the input sentence) and produces the sequence of sentential forms from there until all that remains is the start symbol, which in this grammar is E. The task of the bottom- up parser is to find the specific RHS, the handle, in the sentential form that must be rewritten to get the next (previous) sentential form.
For example, the right sentential form

E + T * id

includes three RHSs, E + T, T, and id. Only one of these is the handle. For example, if the RHS E + T were chosen to be rewritten in this sentential form, the resulting sentential form would be E * id, but E * id is not a legal right sentential form for the given grammar.

The handle of a right sentential form is unique. The task of a bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar. Formally, handle is defined as follows:
Definition: β is the handle of the right sentential form $\gamma = \alpha\beta w$ if and only if

$S =>*_{rm} \alpha A w =>_{rm} \alpha\beta w$

In this definition, $=>_{rm}$ specifies a rightmost derivation step, and $=>*_{rm}$ specifies zero or more rightmost derivation steps.
Definition: β is a phrase of the right sentential form γ if and only if

$S =>* \gamma = \alpha_1 A \alpha_2 =>+ \alpha 1 \beta \alpha 2$

In this definition, => + means one or more derivation steps
Definition: β is a simple phrase of the right sentential form γ if and only if

Downloaded From EnggTree.com

$S \Rightarrow {}^{*}\gamma = \alpha 1 A \alpha 2 \Rightarrow + \alpha 1 \beta \alpha 2$

Shift-Reduce Algorithms:

Bottom- up parsers are often called shift- reduce algorithms, because shift and reduce are the two most common actions they specify. An integral part of every bottom-up parser is a stack. The input to a bottomup parser is the stream of tokens of a program and the output is a sequence of grammar rules. The shift action moves the next input token onto the parser's stack. A reduce action replaces an RHS (the handle) on top of the parser's stack by its corresponding LHS. Every parser for a programming language is a pushdown automaton (PDA), because a PDA is a recognizer for a context-free language.

A PDA is a very simple mathematical machine that scans strings of symbols from left to right. A PDA is so named because it uses a pushdown stack as its memory. A PDA is designed for the purpose of determining whether the string is or is not a sentence in the language. PDA can also produce the information needed to construct a parse tree for the sentence.

LR Parsers:

LR parsers use a relatively small program and a parsing table that is built for a specific programming language.

There are three advantages to LR parsers:

1. They can be built for all programming languages.
2. They can detect syntax errors as soon as it is possible in a left-to-right scan.
3. The LR class of grammars is a proper superset of the class parsable by LL parsers (for example, many left recursive grammars are LR, but none are LL)

The only disadvantage of LR parsing is that it is difficult to produce by hand the parsing table for a given grammar for a complete programming language.

Figure 4.4 shows the structure of an LR parser. The contents of the parse stack for an LR parser have the following form:

$S_0X_1S_1X_2 \ldots X_mS_m$ (top)

where the S's are state symbols and the X's are grammar symbols. An LR parser configuration is a pair of strings (stack, input), with the detailed form

$(S_0X_1S_1X_2S_2 \ldots X_mS_m, a_i\, a_{i+1} \ldots a_n\$)$

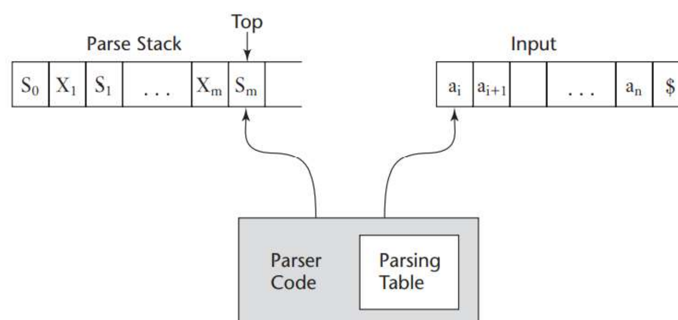


Figure 4.4
The structure of an LR parser

The input string has a dollar sign at its right end. This sign is put there during initialization of the parser.

An LR parsing table has two parts, named ACTION and GOTO. The ACTION part of the table specifies most of what the parser does. It has state symbols as its row labels and the terminal symbols of the grammar as its column labels. Given a current parser state, which is represented by the state symbol on top of the parse stack, and the next symbol (token) of input, the parse table specifies what the parser should do. The two primary parser actions are shift and reduce. Either the parser shifts the next input symbol onto the parse stack or it already has the handle on top of the stack, which it reduces to the LHS

Downloaded From EnggTree.com

of the rule whose RHS is the same as the handle. Two other actions are possible: accept, which means the parser has successfully completed the parse of the input, and error, which means the parser has detected a syntax error.

The rows of the GOTO part of the LR parsing table have state symbols as labels. This part of the table has nonterminals as column labels. The values in the GOTO part of the table indicate which state symbol should be pushed onto the parse stack after a reduction has been completed, which means the handle has been removed from the parse stack and the new nonterminal has been pushed onto the parse stack. The specific symbol is found at the row whose label is the state symbol on top of the parse stack after the handle and its associated state symbols have been removed. The column of the GOTO table that is used is the one with the label, that is the LHS of the rule used in the reduction.

Consider the traditional grammar for arithmetic expressions that follows:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F →id

Abbreviations are used for the actions:

R for reduce and S for shift. R4 means reduce using rule 4; S6 means shift the next symbol of input onto the stack and push state S6 onto the stack. Empty positions in the ACTION table indicate syntax errors.

The initial configuration of an LR parser is (S_0, $a_1 \ldots a_n$\$) The parser actions are informally defined as follows:

1. The Shift process is simple: The next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the ACTION table.

2. For a Reduce action, the handle must be removed from the stack. Because for every grammar symbol on the stack there is a state symbol, the number of symbols removed from the stack is twice the number of symbols in the handle. After removing the handle and its associated state symbols, the LHS of the rule is pushed onto the stack. Finally, the GOTO table is used, with the row label being the symbol that was exposed when the handle and its state symbols were removed from the stack, and the column label being the nonterminal that is the LHS of the rule used in the reduction.

3. When the action is Accept, the parse is complete and no errors were found.

4. When the action is Error, the parser calls an error-handling routine.

Downloaded From EnggTree.com

EnggTree.com

Figure 4.5

The LR parsing table for an arithmetic expression grammar

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Stack	Input	Action
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

Downloaded From EnggTree.com

EnggTree.com

UNIT 2

DATA, DATA TYPES, AND BASIC STATEMENTS

NAMES:

A name is a string of characters used to identify some entity in a program. Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters (_). In the C-based languages, it has to a large extent been replaced by the so-called camel notation, in which all of the words of a multiple-word name except the first are capitalized, as in myStack.

All variable names in PHP must begin with a dollar sign. In Perl, the special character at the beginning of a variable's name, $, @, or %, specifies its type. C-based languages, uppercase and lowercase letters in names are distinct; that is, names in these languages are case sensitive. For example, the following three names are distinct in C++: rose, ROSE, and Rose. In C, the problems of case sensitivity are avoided by the convention that variable names do not include uppercase letters. In Java and C#, however, the problem cannot be escaped because many of the predefined names include both uppercase and lowercase letters. For example, the Java method for converting a string to an integer value is parseInt.

Special Words:

Special words in programming languages are used to make programs more readable by naming actions to be performed. They also are used to separate the syntactic parts of statements and programs. In most languages, special words are classified as reserved words, which means they cannot be redefined by programmers.

A reserved word is a special word of a programming language that cannot be used as a name. There is one potential problem with reserved words: If the language includes a large number of reserved words, the user may have difficulty making up names that are not reserved. The best example of this is COBOL, which has 300 reserved words. Unfortunately, some of the most commonly chosen names by programmers are in the list of reserved words—for example, LENGTH, BOTTOM, DESTINATION, and COUNT.

VARIABLES:

A program variable is an abstraction of a computer memory cell or collection of cells. Programmers often think of variables as names for memory locations, but there is much more to a variable than just a name.

The move from machine languages to assembly languages was largely one of replacing absolute numeric memory addresses for data with names, making programs far more readable and therefore easier to write and maintain. Assembly languages also provided an escape from the problem of manual absolute addressing, because the translator that converted the names to actual addresses also chose those addresses.

A variable can be characterized as a set of attributes: (name, address, value, type, lifetime, and scope).

Downloaded From EnggTree.com

EnggTree.com

Name:

Variable names are the most common names in programs. (Refer NAMES above)

Address:

The address of a variable is the machine memory address with which it is associated. It is possible for the same variable to be associated with different addresses at different times during the execution of the program. For example, if a subprogram has a local variable that is allocated from the run-time stack when the subprogram is called, different calls may result in that variable having different addresses. These are in a sense different instantiations of the same variable.

The address of a variable is sometimes called its l- value, because the address is what is required when the name of a variable appears in the left side of an assignment.

It is possible to have multiple variables that have the same address. When more than one variable name can be used to access the same memory location, the variables are called aliases. Aliasing is a hindrance to readability because it allows a variable to have its value changed by an assignment to a different variable. For example, if variables named total and sum are aliases, any change to the value of total also changes the value of sum and vice versa. Aliasing also makes program verification more difficult.

Type:

The type of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type. For example, the int type in Java specifies a value range of −2147483648 to 2147483647 and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

Value:

The value of a variable is the contents of the memory cell or cells associated with the variable. It is convenient to think of computer memory in terms of abstract cells, rather than physical cells. An abstract memory cell has the size required by the variable with which it is associated. For example, although floating- point values may occupy four physical bytes in a particular implementation of a particular language, a floating- point value is thought of as occupying a single abstract memory cell.

A variable’s value is sometimes called its r- value because it is what is required when the name of the variable appears in the right side of an assignment statement. To access the r-value, the l- value must be determined first.

BINDING:

A binding is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. The time at which a binding takes place is called binding time. Bindings can take place at language design time, language implementation time, compile time, load time, link time, or run time.

Consider the following C++ assignment statement:

Downloaded From EnggTree.com

count = count + 5;

Some of the bindings and their binding times for the parts of this assignment statement are as follows:

• The type of count is bound at compile time.

• The set of possible values of count is bound at compiler design time.

• The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.

• The internal representation of the literal 5 is bound at compiler design time.

• The value of count is bound at execution time with this statement

A complete understanding of the binding times for the attributes of program entities is a prerequisite for understanding the semantics of a programming language.

Binding of Attributes to Variables:

A binding is static if it first occurs before run time begins and remains unchanged throughout program execution. If the binding first occurs during run time or can change in the course of program execution, it is called dynamic. The physical binding of a variable to a storage cell in a virtual memory environment is complex, because the page or segment of the address space in which the cell resides may be moved in and out of memory many times during program execution. In a sense, such variables are bound and unbound repeatedly.

Type Bindings:

Before a variable can be referenced in a program, it must be bound to a data type. The two important aspects of this binding are how the type is specified and when the binding takes place.

Static Type Binding:

An explicit declaration is a statement in a program that lists variable names and specifies that they are a particular type. An implicit declaration is a means of associating variables with types through default conventions, rather than declaration statements. Both explicit and implicit declarations create static bindings to types.

Most widely used programming languages use static type binding.

Implicit variable type binding is done by the language processor, either a compiler or an interpreter. There are several different bases for implicit variable type bindings. The simplest of these is naming conventions. In this case, the compiler or interpreter binds a variable to a type based on the syntactic form of the variable's name.

Some of the problems with implicit declarations can be avoided by requiring names for specific types to begin with particular special characters. For example, in Perl any name that begins with $ is a scalar, which can store either a string or a numeric value. If a name begins with @, it is an array; if it begins with a %, it is a hash structure.3 This creates different namespaces for different type variables. In this scenario, the names @apple and

Downloaded From EnggTree.com

EnggTree.com

%apple are unrelated, because each is from a different namespace. Furthermore, a program reader always knows the type of a variable when reading its name.

Another kind of implicit type declarations uses context. This is sometimes called type inference. For example, in C# a var declaration of a variable must include an initial value, whose type is taken as the type of the variable. Consider the following declarations:

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

The types of sum, total, and name are int, float, and string, respectively.

Dynamic Type Binding:

With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. Instead, the variable is bound to a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment. Such an assignment may also bind the variable to an address and a memory cell, because different type values may require different amounts of storage. Any variable can be assigned any type value. Furthermore, a variable's type can change any number of times during program execution. It is important to realize that the type of a variable whose type is dynamically bound may be temporary.

When the type of a variable is statically bound, the name of the variable can be thought of being bound to a type, in the sense that the type and name of a variable are simultaneously bound. However, when a variable's type is dynamically bound, its name can be thought of as being only temporarily bound to a type. In reality, the names of variables are never bound to types. Names can be bound to variables and variables can be bound to types.

The primary advantage of dynamic binding of variables to types is that it provides more programming flexibility. In Python, Ruby, JavaScript, and PHP, type binding is dynamic. For example, a JavaScript script may contain the following statement:

```
list = [10.2, 3.5];
```

the variable named list, this assignment causes it to become the name of a single-dimensioned array of length 2. If the statement

```
list = 47;
```

list would become the name of a scalar variable.

The option of dynamic type binding was included in C# 2010. A variable can be declared to use dynamic type binding by including the dynamic reserved word in its declaration, as in the following example:

```
dynamic any;
```

Downloaded From EnggTree.com

EnggTree.com

It is similar in that any can be assigned a value of any type, just as if it were declared object. Class members, properties, method parameters, method return values, and local variables can all be declared dynamic.

There are two disadvantages to dynamic type binding. First, it causes programs to be less reliable, because the error-detection capability of the compiler is diminished relative to a compiler for a language with static type bindings. Dynamic type binding allows any variable to be assigned a value of any type. Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type. For example, suppose that in a particular JavaScript program, i and x are currently the names of scalar numeric variables and y is currently the name of an array. Furthermore, suppose that the program needs the assignment statement

i = x; but because of a keying error, it has the assignment statement

i = y;

In JavaScript (or any other language that uses dynamic type binding), no error is detected in this statement by the interpreter— the type of the variable named i is simply changed to an array. But later uses of i will expect it to be a scalar, and correct results will be impossible.

The next disadvantage of dynamic type binding is cost. The cost of implementing dynamic attribute binding is considerable, particularly in execution time. Type checking must be done at run time.

Storage Bindings and Lifetime:

The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called allocation. Deallocation is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory. The lifetime of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell. Storage bindings of variables can be categorized as static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic.

Static Variables:

Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

One advantage of static variables is efficiency. All addressing of static variables can be direct, other kinds of variables often require indirect addressing, which is slower.

One disadvantage of static binding to storage is reduced flexibility; Another disadvantage is that storage cannot be shared among variables.

C and C++ allow programmers to include the static specifier on a variable definition in a function, making the variables it defines static.

Stack-Dynamic Variables:

Stack-dynamic variables are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound. Elaboration of

Downloaded From EnggTree.com

EnggTree.com

such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time.

The advantages of stack-dynamic variables are as follows: recursive subprograms require some form of dynamic local storage so that each active copy of the recursive subprogram has its own version of the local variables.

The disadvantages of stack-dynamic variables are the run- time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required.

Explicit heap- dynamic variables:

Explicit heap- dynamic variables are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These variables, which are allocated from and deallocated to the heap, can only be referenced through pointer or reference variables. The heap is a collection of storage cells whose organization is highly disorganized due to the unpredictability of its use. The pointer or reference variable that is used to access an explicit heap-dynamic variable is created as any other scalar variable.

In C++, the allocation operator, named new, uses a type name as its operand.

As an example of explicit heap-dynamic variables, consider the following C++ code segment: int *intnode; // Create a pointer

intnode = new int; // Create the heap-dynamic variable

. . .

delete intnode; // Deallocate the heap-dynamic variable to which intnode points

Explicit heap-dynamic variables are often used to construct dynamic structures, such as linked lists and trees, that need to grow and/or shrink during execution.

The disadvantages of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the variables, and the complexity of the required storage management implementation.

Implicit Heap-Dynamic Variables:

Implicit heap-dynamic variables are bound to heap storage only when they are assigned values. For example, consider the following JavaScript assignment statement:

highs = [74, 84, 86, 90, 71];

Regardless of whether the variable named highs was previously used in the program or what it was used for, it is now an array of five numeric values.

The advantage of such variables is that they have the highest degree of flexibility.

One disadvantage of implicit heap- dynamic variables is the run- time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others. Another disadvantage is the loss of some error detection by the compiler.

Downloaded From EnggTree.com

TYPE CHECKING:

Type checking is the activity of ensuring that the operands of an operator are of compatible types. A compatible type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler- generated code (or the interpreter) to a legal type. This automatic conversion is called a coercion. For example, if an int variable and a float variable are added in Java, the value of the int variable is coerced to float and a floating-point add is done.

A type error is the application of an operator to an operand of an inappropriate type. For example, in the original version of C, if an int value was passed to a function that expected a float value, a type error would occur.

If all bindings of variables to types are static in a language, then type checking can nearly always be done statically. Dynamic type binding requires type checking at run time, which is called dynamic type checking.

Some languages, such as JavaScript and PHP, because of their dynamic type binding, allow only dynamic type checking. It is better to detect errors at compile time than at run time, because the earlier correction is usually less costly.

Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution.

SCOPE:

The scope of a variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced or assigned in that statement.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable, or how a name is associated with an expression. scope rules determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and their attributes.

A variable is local in a program unit or block if it is declared there. The nonlocal variables of a program unit or block are those that are visible within the program unit or block but are not declared there. Global variables are a special category of nonlocal variables.

Static Scope:

ALGOL 60 introduced the method of binding names to nonlocal variables called static scoping. Static scoping is sometimes called lexical scoping. Static scoping is so named because the scope of a variable can be statically determined— that is, prior to execution.

There are two categories of static-scoped languages: those in which subprograms can be nested, which creates nested static scopes, and those in which subprograms cannot be nested.

When the reader of a program finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared.

Suppose a reference is made to a variable x in subprogram sub1. The correct declaration is found by first searching the declarations of subprogram sub1. If no declaration is found for the variable there, the search continues in the declarations of the subprogram that declared

Downloaded From EnggTree.com

subprogram sub1, which is called its static parent. If a declaration of x is not found there, the search continues to the nextlarger enclosing unit (the unit that declared sub1's parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success. In that case, an undeclared variable error is reported. The static parent of subprogram sub1, and its static parent, and so forth up to and including the largest enclosing subprogram, are called the static ancestors of sub1.

Consider the following JavaScript function, big, in which the two functions sub1 and sub2 are nested: function big()

```
 {
 function sub1()
 {
var x = 7;
sub2();
}
 function sub2()
 {
var y = x;
}
 var x = 3;
 sub1();
}
```

Under static scoping, the reference to the variable x in sub2 is to the x declared in the procedure big. This is true because the search for x begins in the procedure in which the reference occurs, sub2, but no declaration for x is found there. The search continues in the static parent of sub2, big, where the declaration of x is found. The x declared in sub1 is ignored, because it is not in the static ancestry of sub2.

Blocks:

Many languages allow new static scopes to be defined in the midst of executable code. This allows a section of code to have its own local variables whose scope is minimized. Such variables are typically stack dynamic, so their storage is allocated when the section is entered and deallocated when the section is exited. Such a section of code is called a block. Blocks provide the origin of the phrase block-structured language.

Consider the following skeletal C function:

```
void sub()
{
int count;
 . . .
 while (. . .)
{
int count;
 count++;
. . .
}
 . . .
}
```

The reference to count in the while loop is to that loop's local count. In this case, the count of sub is hidden from the code inside the while loop. In general, a declaration for a variable

Downloaded From EnggTree.com

EnggTree.com

effectively hides any declaration of a variable with the same name in a larger enclosing scope. Note that this code is legal in C and C++ but illegal in Java and C#.
Although JavaScript uses static scoping for its nested functions, nonfunction blocks cannot be defined in the language.
Most functional programming languages include a construct that is related to the blocks of the imperative languages, usually named let. These constructs have two parts, the first of which is to bind names to values, usually specified as expressions. The second part is an expression that uses the names defined in the first part.
In Scheme, a let construct is a call to the function LET with the following form:

```
(LET (
(name1 expression1)
. . .
(namen expressionn))
expression
)
```

The semantics of the call to LET is as follows: The first n expressions are evaluated and the values are assigned to the associated names. Then, the final expression is evaluated and the return value of LET is that value.
Consider the following call to LET:

```
(LET (
(top (+ a b))
(bottom (- c d)))
(/ top bottom) )
```

This call computes and returns the value of the expression (a + b) /(c - d).

Global Scope:
Some languages, including C, C++, PHP, JavaScript, and Python, allow a program structure that is a sequence of function definitions, in which variable definitions can appear outside the functions. Definitions outside functions create global variables, which potentially can be visible to those functions.
C and C++ have both declarations and definitions of global data. Declarations specify types and other attributes but do not cause allocation of storage. Definitions specify attributes and cause storage allocation. For a specific global name, a C program can have any number of compatible declarations, but only a single definition.
A declaration of a variable outside function definitions specifies that the variable is defined in a different file.
A global variable that is defined after a function can be made visible in the function by declaring it to be external, as in the following:

```
extern int sum;
```

In C++, a global variable that is hidden by a local with the same name can be accessed using the scope operator (::). For example, if x is a global that is hidden in a function by a local named x, the global could be referenced as ::x.
Any variable that is implicitly declared outside any function is a global variable; variables implicitly declared in functions are local variables.

Global variables can be made visible in functions in their scope in two ways: (1) If the function includes a local variable with the same name as a global, that global can be accessed through the $GLOBALS array, using the name of the global as a string literal subscript, and (2) if there is no local variable in the function with the same name as the global, the global can be made visible by including it in a global declaration statement. Consider the following example:

Downloaded From EnggTree.com

EnggTree.com

```
$day = "Monday";
$month = "January";
 function calendar() {
$day = "Tuesday";
 global $month;
print "local day is $day ";
$gday = $GLOBALS['day'];
print"global day is $gday";
print "global month is $month";
}

calendar();
```

Interpretation of this code produces the following:

```
local day is Tuesday
global day is Monday
global month is January
```

The global variables of JavaScript are very similar to those of PHP, except that there is no way to access a global variable in a function that has declared a local variable with the same name.

Dynamic Scope:

Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the scope can be determined only at run time.

Consider the example:

```
function big() {
function sub1() {
var x = 7;
}
 function sub2() {
 var y = x;
var z = 3;
}
 var x = 3;
}
```

Assume that dynamic-scoping rules apply to nonlocal references. The meaning of the identifier x referenced in sub2 is dynamic— it cannot be determined at compile time. It may reference the variable from either declaration of x, depending on the calling sequence.

One way the correct meaning of x can be determined during execution is to begin the search with the local declarations. This is also the way the process begins with static scoping, but that is where the similarity between the two techniques ends. When the search of local declarations fails, the declarations of the dynamic parent, or calling function, are searched. If a declaration for x is not found there, the search continues in that function's dynamic parent, and so forth, until a declaration for x is found. If none is found in any dynamic ancestor, it is a run-time error.

Consider the two different call sequences for sub2 in the earlier example. First, big calls sub1, which calls sub2. In this case, the search proceeds from the local procedure, sub2, to its caller, sub1, where a declaration for x is found. So, the reference to x in sub2 in this case is to the x declared in sub1. Next, sub2 is called directly from big. In this case, the

Downloaded From EnggTree.com

EnggTree.com

dynamic parent of sub2 is big, and the reference is to the x declared in big. Note that if static scoping were used, in either calling sequence discussed, the reference to x in sub2 would be to big's x.

Scope and Lifetime:

The scope of such a variable is from its declaration to the end of the method. The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

Scope and lifetime are also unrelated when subprogram calls are involved.

Consider the following C++ functions:

```
void printheader() {
. . .
} /* end of printheader */
void compute() {
int sum;
. . .
 printheader();
} /* end of compute */
```

The scope of the variable sum is completely contained within the compute function. It does not extend to the body of the function printheader, although printheader executes in the midst of the execution of compute. However, the lifetime of sum extends over the time during which printheader executes. Whatever storage location sum is bound to before the call to printheader, that binding will continue during and after the execution of printheader.

PRIMITIVE DATATYPE:

A data type defines a collection of data values and a set of predefined operations on those values.

Data types that are not defined in terms of other types are called primitive data types.

Numeric Types:

Some early programming languages only had numeric primitive types. Numeric types still play a central Integer role among the collections of datatypes.

Integer :

The most common primitive numeric data type is integer. The hardware of many computers supports several sizes of integers. For example, Java includes four signed integer sizes: byte, short, int, and long. Some languages, for example, C++ and C#, include unsigned integer types, which are types for integer values without signs. Unsigned types are often used for binary data.

A signed integer value is represented in a computer by a string of bits, with one of the bits (typically the leftmost) representing the sign. Most integer types are supported directly by the hardware.

A negative integer could be stored in sign-magnitude notation, in which the sign bit is set to indicate negative and the remainder of the bit string represents the absolute value of the number.

Most computers now use a notation called twos complement to store negative integers, which is convenient for addition and subtraction. In twos-complement notation, the representation of a negative integer is formed by taking the logical complement of the positive version of the number and adding one. Ones- complement notation is still used by some computers. In ones-complement notation, the negative of an integer is stored as the

Downloaded From EnggTree.com

EnggTree.com

logical complement of its absolute value. Ones- complement notation has the disadvantage that it has two representations of zero.

Floating-Point:

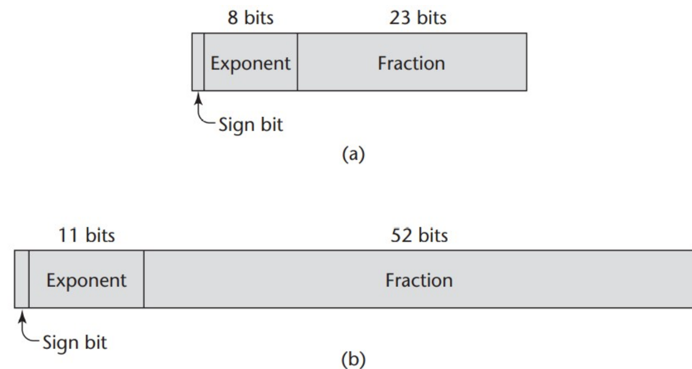
Floating- point data types model real numbers, but the representations are only approximations for many real values. For example, neither of the fundamental numbers pi or e (the base for the natural logarithms) can be correctly represented in floating-point notation. For example, even the value 0.1 in decimal cannot be represented by a finite number of binary digits.

Another problem with floating-point types is the loss of accuracy through arithmetic operations.

Most languages include two floating- point types, often called float and double. The float type is the standard size, usually stored in four bytes of memory. The double type is provided for situations where larger fractional parts and/or a larger range of exponents is needed. Double- precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction. The collection of values that can be represented by a floating-point type is defined in terms of precision and range. Precision is the accuracy of the fractional part of a value, measured as the number of bits. Range is a combination of the range of fractions and, more important, the range of exponents.

Figure 6.1

IEEE floating-point formats: (a) single precision, (b) double precision



Complex:

Some programming languages support a complex data type—for example, Fortran and Python. In Python, the imaginary part of a complex literal is specified by following it with a j or J

for example, (7 + 3j)

Languages that support a complex type include operations for arithmetic on complex values.

Decimal:

Decimal data types store a fixed number of decimal digits, with the implied decimal point at a fixed position in the value.

Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point.

The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful.

Decimal types are stored very much like character strings, using binary codes for the decimal digits. These representations are called binary coded decimal (BCD).

Boolean Types

Boolean types are perhaps the simplest of all types. Their range of values has only two elements: one for true and one for false. Boolean types are often used to represent switches or flags in programs.

Downloaded From EnggTree.com

EnggTree.com

A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

Character Types

Character data are stored in computers as numeric codings. Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters. ISO 8859-1 is another 8-bit character code, but it allows 256 different characters.

CHARACTER STRING TYPES:

A character string type is one in which the values consist of sequences of characters.

Design Issues:

The two most important design issues that are specific to character string types are the following:

- Should strings be a special kind of character array or a primitive type?
- Should strings have static or dynamic length?

Strings and Their Operations:

The most common string operations are assignment, catenation, substring reference, comparison, and pattern matching.

In some languages, pattern matching is supported directly in the language. In others, it is provided by a function or class library.

If strings are not defined as a primitive type, string data is usually stored in arrays of single characters and referenced as such in the language.

For example, consider the following declaration:

```
char str[] = "apples";
```

In this example, str is an array of char elements, specifically apples0, where 0 is the null character. Some of the most commonly used library functions for character strings in C and C++ are strcpy, which moves strings; strcat, which catenates one given string onto another; strcmp, which lexicographically compares (by the order of their character codes) two given strings; and strlen, which returns the number of characters, not counting the null, in the given string. The parameters and return values for most of the string manipulation functions are char pointers that point to arrays of char. Parameters can also be string literals.

The problem is that the functions in this library that move string data do not guard against overflowing the destination. For example, consider the following call to strcpy:

```
strcpy(dest, src);
```

If the length of dest is 20 and the length of src is 50, strcpy will write over the 30 bytes that follow dest. The point is that strcpy does not know the length of dest, so it cannot ensure that the memory following it will not be overwritten.

Downloaded From EnggTree.com

EnggTree.com

A substring reference is a reference to a substring of a given string. The substring references are called slices.

Python includes strings as a primitive type and has operations for substring reference, catenation, indexing to access individual characters, as well as methods for searching and replacement. There is also an operation for character membership in a string. So, even though Python's strings are primitive types, for character and substring references, they act very much like arrays of characters.

Perl, JavaScript, Ruby, and PHP include built-in pattern-matching operations. In these languages, the pattern- matching expressions are somewhat loosely based on mathematical regular expressions. They are often called regular expressions.

Consider the following pattern expression:

/[A-Za-z][A-Za-z\d]+/

This pattern matches (or describes) the typical name form in programming languages. The first character class specifies all letters; the second specifies all letters and digits. The plus operator following the second category specifies that there must be one or more of what is in the category. So, the whole pattern matches strings that begin with a letter, followed by one or more letters or digits.

String Length Options:

There are several design choices regarding the length of string values.

First, the length can be static and set when the string is created. Such a string is called a static length string.

The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable's definition, as exemplified by the strings in C and the C-style strings of C++. These are called limited dynamic length strings.

The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library. These are called dynamic length strings.

Implementation of Character String Types:

Character string types could be supported directly in hardware; but in most cases, software is used to implement string storage, retrieval, and manipulation.

A descriptor for a static character string type, which is required only during compilation, has three fields. The first field of every descriptor is the name of the type. In the case of static character strings, the second field is the type's length (in characters). The third field is the address of the first character.

Limited dynamic strings require a run-time descriptor to store both the fixed maximum length and the current length

Downloaded From EnggTree.com

Figure 6.2

Compile-time descriptor for static strings

Static string
Length
Address

Figure 6.3

Run-time descriptor for limited dynamic strings

Limited dynamic string
Maximum length
Current length
Address

Dynamic length strings require more complex storage management. The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically.

There are three approaches to supporting the dynamic allocation and deallocation that is required for dynamic length strings.

First, strings can be stored in a linked list, so that when a string grows, the newly required cells can come from anywhere in the heap. The drawbacks to this method are the extra storage occupied by the links in the list representation and the necessary complexity of string operations.

The second approach is to store strings as arrays of pointers to individual characters allocated in the heap. This method still uses extra memory, but string processing can be faster than with the linked-list approach.

The third alternative is to store complete strings in adjacent storage cells. The problem with this method arises when a string grows: How can storage that is adjacent to the existing cells continue to be allocated for the string variable? Frequently, such storage is not available.

The linked-list method requires more storage, the associated allocation and deallocation processes are simple.

ARRAY TYPES:

An array is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element. The individual data elements of an array are of the same type. References to individual array elements are specified using subscript expressions. If any of the subscript expressions in a reference include variables, then the reference will require an additional run-time calculation to determine the address of the memory location being referenced.

Arrays and Indices:

Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate name, and the second part is a possibly dynamic selector consisting of one or more items known as subscripts or indices. If all of the subscripts in a reference are constants, the selector is static; otherwise, it is dynamic. arrays are sometimes called finite mappings. Symbolically, this mapping can be shown as

array_name(subscript_value_list) → element

The syntax of array references is fairly universal: The array name is followed by the list of subscripts, which is surrounded by either parentheses or brackets. Two distinct types are

Downloaded From EnggTree.com

EnggTree.com

involved in an array type: the element type and the type of the subscripts. The type of the subscripts is often integer.

Subscript Bindings and Array Categories:

The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound. In some languages, the lower bound of the subscript range is implicit. For example, in the C-based languages, the lower bound of all subscript ranges is fixed at 0.

There are four categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated.

A static array is one in which the subscript ranges are statically bound and storage allocation is static (done before run time). The advantage of static arrays is efficiency: No dynamic allocation or deallocation is required. The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

A fixed stack-dynamic array is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution. The advantage of fixed stack-dynamic arrays over static arrays is space efficiency. A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time. The disadvantage is the required allocation and deallocation time.

A fixed heap-dynamic array is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack. The advantage of fixed heap-dynamic arrays is flexibility—the array's size always fits the problem. The disadvantage is allocation time from the heap, which is longer than allocation time from the stack.

A heap- dynamic array is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime. The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes. The disadvantage is that allocation and deallocation take longer and may happen many times during execution of the program. Examples of the four categories are given in the following paragraphs.

Arrays declared in C and C++ functions that include the static modifier are static. Arrays that are declared in C and C++ functions without the static specifier are examples of fixed stack-dynamic arrays.

C and C++ also provide fixed heap-dynamic arrays. The standard C library functions malloc and free, which are general heap allocation and deallocation operations, can be used for C arrays. C++ uses the operators new and delete to manage heap storage.

Objects are created without any elements, as in

Downloaded From EnggTree.com

EnggTree.com

List <String>stringList = new List <String>();

Elements are added to this object with the Add method, as in

stringList.Add("Michael");

Array Initialization:

Consider the following C declaration:

int list [] = {4, 5, 7, 83};

The array list is created and initialized with the values 4, 5, 7, and 83. The compiler also sets the length of the array.

These arrays can be initialized to string constants, as in

char name [] = "freddie";

The array name will have eight elements, because all strings are terminated with a null character (zero), which is implicitly supplied by the system for string constants. Arrays of strings in C and C++ can also be initialized with string literals. For example,

char *names [] = {"Bob", "Jake", "Darcie"};

Array Operations:

An array operation is one that operates on an array as a unit. The most common array operations are assignment, catenation, comparison for equality and inequality, and slices.

In APL, the four basic arithmetic operations are defined for vectors (singledimensioned arrays) and matrices, as well as scalar operands. For example,

A + B is a valid expression, whether A and B are scalar variables, vectors, or matrices. APL includes a collection of unary operators for vectors and matrices, some of which are as follows (where V is a vector and M is a matrix):

ϕV reverses the elements of V

ϕM reverses the columns of M

θM reverses the rows of M

ØM transposes M (its rows become its columns and vice versa)

÷M inverts M

APL also includes several special operators that take other operators as operands. One of these is the inner product operator, which is specified with a period (.). It takes two operands, which are binary operators. For example, +.×

For example, if A and B are vectors,

A × B is the mathematical inner product of A and B (a vector of the products of the corresponding elements of A and B). The statement

Downloaded From EnggTree.com

EnggTree.com

A +.× B is the sum of the inner product of A and B. If A and B are matrices, this expression specifies the matrix multiplication of A and B

Rectangular and Jagged Arrays:

A rectangular array is a multidimensional array in which all of the rows have the same number of elements and all of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly. A jagged array is one in which the lengths of the rows need not be the same. For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements

Slice:

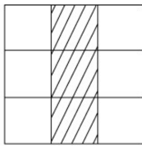
A slice of an array is some substructure of that array. For example, if A is a matrix, then the first row of A is one possible slice, as are the last row and the first column. Slice is not a new data type, it is a mechanism for referencing part of an array as a unit.

In Fortran 95,

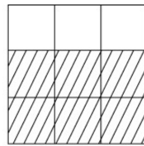
Integer, Dimension (10) :: Vector

Integer, Dimension (3, 3) :: Mat

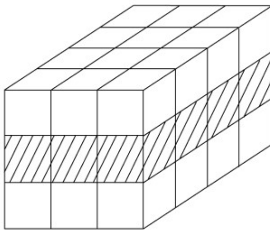
Integer, Dimension (3, 3) :: Cube



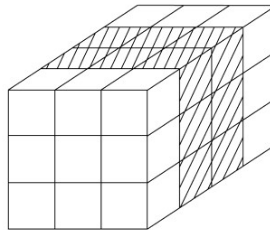
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

Ruby supports slices with the slice method. . A single integer expression parameter is interpreted as a subscript, in which case slice returns the element with the given subscript. If slice is given two integer expression parameters, the first is interpreted as a beginning subscript and the second is interpreted as the number of elements in the slice. For example, suppose list is defined as follows: list = [2, 4, 6, 8, 10] list.slice(2, 2) returns [6, 8]. The third parameter form for slice is a range, which has the form of an integer expression, two periods, and a second integer expression.

Implementation of Array Types

Implementing arrays requires considerably more compile-time effort than does implementing primitive types. The code to allow accessing of array elements must be generated at compile

Downloaded From EnggTree.com

EnggTree.com

time. At run time, this code must be executed to produce element addresses. There is no way to precompute the address to be accessed by a reference such as

list[k]

A single-dimensioned array is implemented as a list of adjacent memory cells. Suppose the array list is defined to have a subscript range lower bound of 0. The access function for list is often of the form address(list[k]) = address(list[0]) + k * element_size where the first operand of the addition is the constant part of the access function, and the second is the variable part.

The generalization of this access function for an arbitrary lower bound is address(list[k]) = address(list[lower_bound]) + ((k -lower_bound) * element_size)

The compile-time descriptor for single-dimensioned arrays can have the form shown in Figure 6.4. The descriptor includes information required to construct the access function. If run-time checking of index ranges is not done and the attributes are all static, then only the access function is required during execution; no descriptor is needed. If run-time checking of index ranges is done, then those index ranges may need to be stored in a run-time descriptor. If the subscript ranges of a particular array type are static, then the ranges may be incorporated into the code that does the checking, thus eliminating the need for the run-time descriptor. If any of the descriptor entries are dynamically bound, then those parts of the descriptor must be maintained at run time.

Figure 6.4

Compile-time descriptor for single-dimensioned arrays

Array
Element type
Index type
Index lower bound
Index upper bound
Address

There are two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order (not used in any widely used language). In row major order, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth.

location(a[i,j]) = address of a[0, 0] + ((((number of rows above the ith row) * (size of a row)) + (number of elements left of the jth column)) * element size)

Figure 6.6

A compile-time descriptor for a multidimensional array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

Downloaded From EnggTree.com

EnggTree.com

ASSOCIATIVE ARRAY:

An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys. In the case of non-associative arrays, the indices never need to be stored. Each element of an associative array is in fact a pair of entities, a key and a value. Associative arrays are also supported directly by Python, Ruby, and Lua and by the standard class libraries of Java, C++, C#, and F#. The only design issue that is specific for associative arrays is the form of references to their elements.

Structure and Operations:

In Perl, associative arrays are called hashes, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (%). Each hash element consists of two parts: a key, which is a string, and a value, which is a scalar (number, string, or reference). Hashes can be set to literal values with the assignment statement, as in

%salaries = ("Gary" => 75000, "Perry" => 57000, "Mary" => 55750, "Cedric" => 47850);

Individual element values are referenced using notation that is similar to that used for Perl arrays. The key value is placed in braces and the hash name is replaced by a scalar variable name. Scalar variable names begin with dollar signs ($).So, an assignment of 58850 to the element of %salaries with the key "Perry" would appear as follows: $salaries{"Perry"} = 58850;

An element can be removed from the hash with the delete operator, as in the following:

delete $salaries{"Gary"};

The entire hash can be emptied by assigning the empty literal to it, as in the following:

@salaries = ();

The size of a Perl hash is dynamic: It grows when an element is added and shrinks when an element is deleted, and also when it is emptied by assignment.

The exists operator returns true or false, depending on whether its operand key is an element in the hash. For example,

if (exists $salaries{"Shelly"}) . . .

Python's associative arrays, which are called dictionaries, are similar to those of Perl, except the values are all references to objects. The associative arrays supported by Ruby are similar to those of Python, except that the keys can be any object,rather than just strings.

PHP's arrays are both normal arrays and associative arrays. They can be treated as either. The language provides functions that allow both indexed and hashed access to elements. An array can have elements that are created with simple numeric indices and elements that are created with string hash keys.

An associative array is much better than an array if searches of the elements are required, because the implicit hashing operation used to access elements is very efficient. Furthermore, associative arrays are ideal when the data to be stored is paired, as with employee names and

Downloaded From EnggTree.com

EnggTree.com

their salaries. On the other hand, if every element of a list must be processed, it is more efficient to use an array.

Implementing Associative Arrays:

The implementation of Perl's associative arrays is optimized for fast lookups, but it also provides relatively fast reorganization when array growth requires it. When an associative array must be expanded beyond its initial size, the hash function need not be changed; rather, more bits of the hash value are used.

RECORD TYPE:

A record is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.

There is a need in programs to model a collection of data in which the individual elements are not of the same type or size. For example, information about a college student might include name, student number, grade point average, and so forth. A data type for such a collection might use a character string for the name, an integer for the student number, a floating-point for the grade point average, and so forth. Records are designed for this kind of need.

Records and heterogeneous arrays are not same. The elements of a heterogeneous array are all references to data objects that reside in scattered locations, often on the heap. The elements of a record are of potentially different sizes and reside in adjacent memory locations. The fundamental difference between a record and an array is that record elements, or fields, are not referenced by indices. Instead, the fields are named with identifiers, and references to the fields are made using these identifiers.

In C, C++, and C#, records are supported with the struct data type.

Definitions of Records:

The COBOL form of a record declaration,

```
01 EMPLOYEE-RECORD.
        02 EMPLOYEE-NAME.
                05 FIRST PICTURE IS X(20).
                05 Middle PICTURE IS X(10).
                05 LAST PICTURE IS X(20).
        02 HOURLY-RATE PICTURE IS 99V99.
```

The EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field. The numerals 01, 02, and 05 that begin the lines of the record declaration are level numbers, which indicate by their relative values the hierarchical structure of the record. The PICTURE clauses show the formats of the field storage locations, with X(20) specifying 20 alphanumeric characters and 99V99 specifying four decimal digits with the decimal point in the middle.

consider the following declaration:

Downloaded From EnggTree.com

EnggTree.com

employee.name = "Freddie"

employee.hourlyRate = 13.20

These assignment statements create a table (record) named employee with two elements (fields) named name and hourlyRate, both initialized.

References to Record Fields:

References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and its enclosing records. COBOL field references have the form

field_name OF record_name_1 OF . . . OF record_name_n

where the first record named is the smallest or innermost record that contains the field. The next record name in the sequence is that of the record that contains the previous record, and so forth.

Example: Middle OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

Most of the other languages use dot notation for field references. Names in dot notation have the opposite order of COBOL references: They use the name of the largest enclosing record first and the field name last.

Employee_Record.Employee_Name.Middle

A fully qualified reference to a record field is one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference. COBOL allows elliptical references to record fields. In an elliptical reference, the field is named, but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous in the referencing environment. For example,

FIRST, FIRST OF EMPLOYEE- NAME, and FIRST OF EMPLOYEERECORD are elliptical references.

Evaluation:

Arrays are used when all the data values have the same type and/or are processed in the same way. This processing is easily done when there is a systematic way of sequencing through the structure. Such processing is well supported by using dynamic subscripting as the addressing method.

Records are used when the collection of data values is heterogeneous and the different fields are not processed in the same way. Also, the fields of a record often need not be processed in a particular order.

Downloaded From EnggTree.com

EnggTree.com

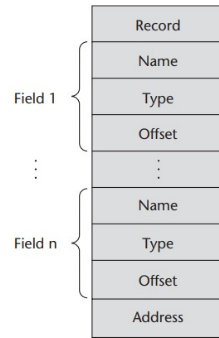


Figure 6.7
A compile-time descriptor for a record

Implementation of Record Types:

The fields of records are stored in adjacent memory locations. But the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records.

UNION TYPES:

A union is a type whose variables may store different type values at different times during program execution.

Discriminated Versus Free Unions:

C and C++ provide union constructs in which there is no language support for type checking. In C and C++, the union construct is used to specify union structures. The unions in these languages are called free unions, because programmers are allowed complete freedom from type checking in their use. For example, consider the following C union:

```
union flexType {
 int intEl;
 float floatEl;
};
union flexType el1;
float x;
. . .
el1.intEl = 27;
x = el1.floatEl;
```

This last assignment is not type checked, because the system cannot determine the current type of the current value of el1, so it assigns the bit string representation of 27 to the float variable x, which of course is nonsense. Type checking of unions requires that each union construct include a type indicator. Such an indicator is called a tag, or discriminant, and a union with a discriminant is called a discriminated union. The first language to provide discriminated unions was ALGOL 68. They are now supported by ML, Haskell, and F#

Downloaded From EnggTree.com

EnggTree.com

Unions in F#:

A union is declared in F# with a type statement using OR operators (|) to define the components. For example, we could have the following:

```
type intReal =
    | IntValue of int
    | RealValue of float;;
```

In this example, intReal is the union type. IntValue and RealValue are constructors. Values of type intReal can be created using the constructors as if they were a function, as in the following examples:

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

Accessing the value of a union is done with a pattern-matching structure. Pattern matching in F# is specified with the match reserved word. The general form of the construct is as follows:

match pattern with

| expression_list$_1$ - > expression$_1$

| . . .

| expression_list$_n$ - > expression$_n$

The pattern can be any data type. The expression list can include wild card characters (_) or be solely a wild card character. For example, consider the following match construct:

```
 let a = 7;;
let b = "grape";;
 let x = match (a, b) with
| 4, "apple" -> apple
| _, "grape" -> grape
| _ -> fruit;;
```

To display the type of the intReal union, the following function could be used:

```
let printType value = match value with
| IntValue value -> printfn "It is an integer"
| RealValue value -> printfn "It is a float";;
```

The following lines show calls to this function and the output:

```
printType ir1;; It is an integer
printType ir2;; It is a float
```

Downloaded From EnggTree.com

EnggTree.com

Evaluation:

Unions are potentially unsafe constructs in some languages. They are one of the reasons why C and C++ are not strongly typed: These languages do not allow type checking of references to their unions. On the other hand, unions can be safely used, as in their design in ML, Haskell, and F#.

POINTER AND REFERENCE TYPES:

A pointer type is one in which the variables have a range of values that consists of memory addresses and a special value, nil.

Pointers are designed for two distinct kinds of uses. First, pointers provide some of the power of indirect addressing, which is frequently used in assembly language programming. Second, pointers provide a way to manage dynamic storage. A pointer can be used to access a location in an area where storage is dynamically allocated called a heap.

Variables that are dynamically allocated from the heap are called heapdynamic variables. They often do not have identifiers associated with them and thus can be referenced only by pointer or reference type variables. Variables without names are called anonymous variables.

Pointers, unlike arrays and records, are not structured types, although they are defined using a type operator (* in C and C++). Furthermore, they are also different from scalar variables because they are used to reference some other variable, rather than being used to store data. These two categories of variables are called reference types and value types, respectively

Pointer Operations:

A pointer type usually include two fundamental pointer operations: assignment and dereferencing. The first operation sets a pointer variable's value to some useful address.

An occurrence of a pointer variable in an expression can be interpreted in two distinct ways. First, it could be interpreted as a reference to the contents of the memory cell to which it is bound, which in the case of a pointer is an address. However, the pointer also could be interpreted as a reference to the value in the memory cell pointed to by the memory cell to which the pointer variable is bound. In this case, the pointer is interpreted as an indirect reference. The former case is a normal pointer reference; the latter is the result of dereferencing the pointer. Dereferencing, is the second fundamental pointer operation.

Dereferencing of pointers can be either explicit or implicit. In many contemporary languages, it occurs only when explicitly specified. In C++, it is explicitly specified with the asterisk (*) as a prefix unary operator.

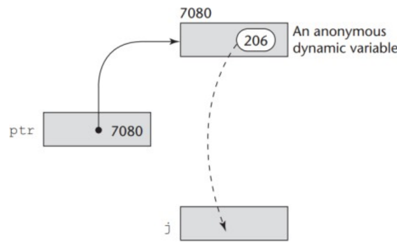
Consider the following example of dereferencing: If ptr is a pointer variable with the value 7080 and the cell whose address is 7080 has the value 206, then the assignment

j = *ptr

sets j to 206.

Downloaded From EnggTree.com

EnggTree.com



When pointers point to records, In C and C++, there are two ways a pointer to a record can be used to reference a field in that record. If a pointer variable p points to a record with a field named age, (*p).age can be used to refer to that field. The operator ->, when used between a pointer to a struct and a field of that struct, combines dereferencing and field reference. For example, the expression

p -> age is equivalent to (*p).age.

Pointer Problems:

The first high- level programming language to include pointer variables was PL/I, in which pointers could be used to refer to both heap-dynamic variables and other program variables.

Dangling Pointers:

A dangling pointer, or dangling reference, is a pointer that contains the address of a heap-dynamic variable that has been deallocated. Dangling pointers are dangerous for several reasons. First, the location being pointed to may have been reallocated to some new heap-dynamic variable. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid. Even if the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value.

The following sequence of operations creates a dangling pointer in many languages:

1. A new heap-dynamic variable is created and pointer p1 is set to point at it.

2. Pointer p2 is assigned p1's value.

3. The heap-dynamic variable pointed to by p1 is explicitly deallocated (possibly setting p1 to nil), but p2 is not changed by the operation. p2 is now a dangling pointer. If the deallocation operation did not change p1, both p1 and p2 would be dangling. (Of course, this is a problem of aliasing—p1 and p2 are aliases.)

For example, in C++ we could have the following:

```
int * arrayPtr1;

int * arrayPtr2 = new int[100];

arrayPtr1 = arrayPtr2;

delete [] arrayPtr2;

// Now, arrayPtr1 is dangling, because the heap storage

// to which it was pointing has been deallocated.
```

Downloaded From EnggTree.com

EnggTree.com

Lost Heap-Dynamic Variables:

A lost heap- dynamic variable is an allocated heap- dynamic variable that is no longer accessible to the user program. Such variables are often called garbage, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program. Lost heap-dynamic variables are most often created by the following sequence of operations:

1. Pointer p1 is set to point to a newly created heap-dynamic variable.

2. p1 is later set to point to another newly created heapdynamic variable.

The first heap-dynamic variable is now inaccessible, or lost. This is sometimes called memory leakage.

Pointers in C and C++:

In C and C++, pointers can be used in the same ways as addresses are used in assembly languages. In C and C++, the asterisk (*) denotes the dereferencing operation and the ampersand (&) denotes the operator for producing the address of a variable. For example, consider the following code:

```
int *ptr;
int count, init;
 . . .
ptr = &init;
count = *ptr;
```

The assignment to the variable ptr sets it to the address of init. The assignment to count dereferences ptr to produce the value at init, which is then assigned to count. So, the effect of the two assignment statements is to assign the value of init to count. Notice that the declaration of a pointer specifies its domain type.

In C and C++, all arrays use zero as the lower bound of their subscript ranges, and array names without subscripts always refer to the address of the first element.

Consider the following declarations:

```
int list [10];
 int *ptr;
```

Now consider the assignment

```
ptr = list;
```

This assigns the address of list[0] to ptr. Given this assignment, the following are true:

• *(ptr + 1) is equivalent to list[1].

Downloaded From EnggTree.com

EnggTree.com

- *(ptr + index) is equivalent to list[index].
- ptr[index] is equivalent to list[index].

It is clear from these statements that the pointer operations include the same scaling that is used in indexing operations.

Reference Types:

A reference type variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory.

Reference type variables are specified in definitions by preceding their names with ampersands (&). For example,

```
int result = 0;
 int &ref_result = result;
 . . .
ref_result = 100;
```

When used as formal parameters in function definitions, C++ reference types provide for two-way communication between the caller function and the called function. This is not possible with nonpointer primitive parameter types, because C++ parameters are passed by value. Passing a pointer as a parameter accomplishes the same two- way communication, but pointer formal parameters require explicit dereferencing, making the code less readable and less safe. Reference parameters are referenced in the called function exactly as are other parameters. The calling function need not specify that a parameter whose corresponding formal parameter is a reference type is anything unusual. The compiler passes addresses, rather than values, to reference parameters

Solutions to the Dangling-Pointer Problem:

There have been several proposed solutions to the dangling-pointer problem. Among these are **tombstones** (Lomet, 1975), in which every heap- dynamic variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable. The actual pointer variable points only at tombstones and never to heap-dynamic variables. When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic variable no longer exists. This approach prevents a pointer from ever pointing to a deallocated variable.

Tombstones are costly in both time and space. Because tombstones are never deallocated, their storage is never reclaimed. Every access to a heap dynamic variable through a tombstone requires one more level of indirection, which requires an additional machine cycle on most computers.

An alternative to tombstones is the **locks-and- keys approach.** In this compiler, pointer values are represented as ordered pairs (key, address), where the key is an integer value. When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in

Downloaded From EnggTree.com

EnggTree.com

the call to new. Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable. If they match, the access is legal; otherwise the access is treated as a run-time error. When a heap-dynamic variable is deallocated with dispose, its lock value is cleared to an illegal lock value.

The best solution to the dangling- pointer problem is to take deallocation of heap-dynamic variables out of the hands of programmers. If programs cannot explicitly deallocate heap-dynamic variables, there will be no dangling pointers.

ARITHMETIC EXPRESSIONS:

Automatic evaluation of arithmetic expressions similar to those found in matematics, science, and engineering was one of the primary goals of the first high-level programming languages. An operator can be unary, meaning it has a single operand, binary, meaning it has two operands, or ternary, meaning it has three operands. In most programming languages, binary operators are infix, which means they appear between their operands. One exception is Perl, which has some operators that are prefix, which means they precede their operands.

Operator Evaluation Order:

The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

Precedence:

The value of an expression depends at least in part on the order of evaluation of the operators in the expression. Consider the following expression:

a + b * c

Suppose the variables a, b, and c have the values 3, 4, and 5, respectively. If evaluated left to right (the addition first and then the multiplication), the result is 35. If evaluated right to left, the result is 23.

The operator precedence rules for expression evaluation partially define the order in which the operators of different precedence levels are evaluated. Exponentiation has the highest precedence (when it is provided by the language), followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level.

Many languages also include unary versions of addition and subtraction. Unary addition is called the identity operator because it usually has no associated operation and thus has no effect on its operand.

The unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is parenthesized to prevent it from being next to another operator. For example,

a + (- b) * c is legal,

but a + - b * c usually is not.

Downloaded From EnggTree.com

EnggTree.com

The precedences of the arithmetic operators of Ruby and the C-based languages are as follows:

	Ruby	*C-Based Languages*
Highest	**	postfix ++, --
	unary +, -	prefix ++, --, unary +, -
	*, /, %	*, /, %
Lowest	binary +, -	binary +, -

Associativity:

When an expression contains two adjacent occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the associativity rules of the language. An operator can have either left or right associativity, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first.

In the Java expression a - b + c the left operator is evaluated first

Exponentiation in Fortran and Ruby is right associative, so in the expression

A ** B ** C

the right operator is evaluated first.

In Visual Basic, the exponentiation operator, ^, is left associative.

The associativity rules for a few common languages are given here:

Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +

Parentheses:

Programmers can alter the precedence and associativity rules by placing parentheses in expressions. A parenthesized part of an expression has precedence over its adjacent unparenthesized parts. For example, although multiplication has precedence over addition, in the expression

(A + B) * C

The disadvantage of this scheme is that it makes writing expressions more tedious.

Ruby:

Ruby is a pure object-oriented language, which means, among other things, that every data value, including literals, is an object. For example, the expression a + b is a call to the + method of the object referenced by a, passing the object referenced by b as a parameter.

Expressions in Lisp:

In Lisp, the subprograms must be explicitly called. For example, to specify the C expression

Downloaded From EnggTree.com

EnggTree.com

a + b * c in Lisp, one must write the following expression:

(+ a (* b c))

In this expression, + and * are the names of functions.

Conditional Expressions:

if-then-else statements can be used to perform a conditional expression assignment. For example, consider

```
if (count == 0)
average = 0;
else
average = sum / count;
```

assignment statement using a conditional expression, which has the following form:

expression_1 ? expression_2 : expression_3

where expression_1 is interpreted as a Boolean expression. If expression_1 evaluates to true, the value of the whole expression is the value of expression_2; otherwise, it is the value of expression_3.

average = (count == 0) ? 0 : sum / count;

Operand Evaluation Order:

Variables in expressions are evaluated by fetching their values from memory. Constants are sometimes evaluated the same way. In other cases, a constant may be part of the machine language instruction and not require a memory fetch. If an operand is a parenthesized expression, all of the operators it contains must be evaluated before its value can be used as an operand. If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant.

Side Effects:

A side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable.

Consider the following expression:

a + fun(a)

If fun does not have the side effect of changing a, then the order of evaluation of the two operands, a and fun(a), has no effect on the value of the expression.

Suppose we have the following:

a = 10; b = a + fun(a);

Then, if the value of a is fetched first (in the expression evaluation process), its value is 10 and the value of the expression is 20. But if the second operand is evaluated first, then the value of the first operand is 20 and the value of the expression is 30.

Downloaded From EnggTree.com

EnggTree.com

Referential Transparency and Side Effects:

A program has the property of referential transparency if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.

result1 = (fun(a) + b) / (fun(a) - c);

temp = fun(a);

result2 = (temp + b) / (temp - c);

If the function fun has no side effects, result1 and result2 will be equal, because the expressions assigned to them are equivalent. However, suppose fun has the side effect of adding 1 to either b or c. Then result1 would not be equal to result2. So, that side effect violates the referential transparency of the program in which the code appears.

There are several advantages to referentially transparent programs. The most important of these is that the semantics of such programs is much easier to understand than the semantics of programs that are not referentially transparent.

Overloaded Operators:

Arithmetic operators are often used for more than one purpose. For example, + usually is used to specify integer addition and floating-point addition. This multiple use of an operator is called operator overloading.

consider the use of the ampersand (&) in C++. As a binary operator, it specifies a bitwise logical AND operation. As a unary operator, however, its meaning is totally different. As a unary operator with a variable as its operand, the expression value is the address of that variable. In this case, the ampersand is called the address-of operator. For example, the execution of x = &y; causes the address of y to be placed in x.

There are two problems with this multiple use of the ampersand.

First, using the same symbol for two completely unrelated operations is detrimental to readability. Second, the simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler. The problem is only that the compiler cannot tell if the operator is meant to be binary or unary.

Suppose a user wants to define the * operator between a scalar integer and an integer array to mean that each element of the array is to be multiplied by the scalar. Such an operator could be defined by writing a function subprogram named * that performs this new operation. The compiler will choose the correct meaning when an overloaded operator is specified, based on the types of the operands, as with language-defined overloaded operators.

For example, if + and * are overloaded for a matrix abstract data type and A, B, C, and D are variables of that type, then

A * B + C * D can be used instead of

MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))

Downloaded From EnggTree.com

EnggTree.com

C++ has a few operators that cannot be overloaded. Among these are the class or structure member operator (.) and the scope resolution operator (::).

TYPE CONVERSIONS:

Type conversions are either narrowing or widening. A narrowing conversion converts a value to a type that cannot store even approximations of all of the values of the original type. For example, converting a double to a float in Java is a narrowing conversion, because the range of double is much larger than that of float. A widening conversion converts a value to a type that can include at least approximations of all of the values of the original type. For example, converting an int to a float in Java is a widening conversion.

Narrowing conversions are not always safe—sometimes the magnitude of the converted value is changed in the process. Widening conversions are usually safe. Type conversions can be either explicit or implicit.

Coercion in Expressions:

One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types. Languages that allow such expressions, which are called mixed-mode expressions, must define conventions for implicit operand type conversions because computers do not have binary operations that take operands of different types.

coercion was defined as an implicit type conversion that is initiated by the compiler or runtime system. Type conversions explicitly requested by the programmer are referred to as explicit conversions, or casts.

When the two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced and generate the code for that coercion.

consider the following Java code:

```
int a;

float b, c, d;

. . .

d = b * a;
```

Assume that the second operand of the multiplication operator was supposed to be c, but because of a keying error it was typed as a. Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. It would simply insert code to coerce the value of the int operand, a, to float. If mixed-mode expressions were not legal in Java, this keying error would have been detected by the compiler as a type error.

Explicit Type Conversion:

Explicit type conversions are called casts. To specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

Downloaded From EnggTree.com

EnggTree.com

(int) angle

One of the reasons for the parentheses around the type name in these conversions is that the first of these languages, C, has several two-word type names, such as long int.

Errors in Expressions:

A number of errors can occur during expression evaluation. The most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored. This is called overflow or underflow, depending on whether the result was too large or too small. One limitation of arithmetic is that division by zero is disallowed.

Floating-point overflow, underflow, and division by zero are examples of run-time errors, which are sometimes called exceptions.

RELATIONAL AND BOOLEAN EXPRESSIONS:

Relational Expressions:

A relational operator is an operator that compares the values of its two operands. A relational expression has two operands and one relational operator. The types of the operands that can be used for relational operators are numeric types, strings, and enumeration types. The syntax of the relational operators for equality and inequality differs among some programming languages. For example, for inequality, the C-based languages use !=, Lua uses ~=, Fortran 95+ uses .NET or <>, and ML and F# use <>.

JavaScript and PHP have two additional relational operators, === and !==. These are similar to their relatives, == and !=

Boolean Expressions:

Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators. The operators usually include those for the AND, OR, and NOT operations, and sometimes for exclusive OR and equivalence. Boolean operators usually take only Boolean operands (Boolean variables, Boolean literals, or relational expressions) and produce Boolean values.

The precedence of the arithmetic, relational, and Boolean operators in the C-based languages is as follows:

Highest	postfix ++, --
	unary +, unary -, prefix ++, --, !
	*, /, %
	binary +, binary -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

ASSIGNMENT STATEMENTS:

It provides the mechanism by which the user can dynamically change the bindings of values to variables.

Downloaded From EnggTree.com

EnggTree.com

Simple Assignments:

All programming languages use the equal sign for the assignment operator. ALGOL 60 pioneered the use of := as the assignment operator, which avoids the confusion of assignment with equality. Ada also uses this assignment operator.

Conditional Targets:

Perl allows conditional targets on assignment statements. For example,

consider

($flag ? $count1 : $count2) = 0;

which is equivalent to

if ($flag)

{ $count1 = 0; }

else { $count2 = 0; }

Compound Assignment Operators:

A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment. The form of assignment that can be abbreviated with this technique has the destination variable also appearing as the first operand in the expression on the right side, as in

a = a + b; a+=b

The syntax of these assignment operators is the catenation of the desired binary operator to the = operator. For example,

sum += value;

is equivalent to

sum = sum + value;

The languages that support compound assignment operators have versions for most of their binary operators

Unary Assignment Operators:

The C-based languages, Perl, and JavaScript include two special unary arithmetic operators that are actually abbreviated assignments. They combine increment and decrement operations with assignment. The operators ++ for increment and --for decrement can be used either in expressions or to form stand-alone single-operator assignment statements. They can appear either as prefix operators, meaning that they precede the operands, or as postfix operators, meaning that they follow the operands. In the assignment statement

sum = ++ count;

the value of count is incremented by 1 and then assigned to sum.

Downloaded From EnggTree.com

EnggTree.com

This operation could also be stated as

count = count + 1;

sum = count;

If the same operator is used as a postfix operator, as in

sum = count ++;

the assignment of the value of count to sum occurs first; then count is incremented. The effect is the same as that of the two statements

sum = count;

count = count + 1;

An example of the use of the unary increment operator to form a complete assignment statement is count ++; which simply increments count. It does not look like an assignment, but it certainly is one. It is equivalent to the statement

count = count + 1;

Assignment as an Expression:

In the C-based languages, Perl, and JavaScript, the assignment statement produces a result, which is the same as the value assigned to the target. It can therefore be used as an expression and as an operand in other expressions.

For example, the expression

a = b + (c = d / b) - 1

denotes the instructions

Assign d / b to c

Assign b + c to temp

Assign temp - 1 to a

Note that the treatment of the assignment operator as any other binary operator allows the effect of multiple-target assignments, such as

sum = count = 0;

in which count is first assigned the zero, and then count's value is assigned to sum. This form of multiple-target assignments is also legal in Python.

There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors. In particular, if we type

if (x = y) ...

instead of

if (x == y) ...

Downloaded From EnggTree.com

EnggTree.com

which is an easily made mistake, it is not detectable as an error by the compiler.

Multiple Assignments:

Several recent programming languages, including Perl, Ruby, and Lua, provide multiple-target, multiple-source assignment statements. For example, in Perl one can write

($first, $second, $third) = (20, 40, 60);

The semantics is that 20 is assigned to $first, 40 is assigned to $second, and 60 is assigned to $third. If the values of two variables must be interchanged, this can be done with a single assignment, as with

($first, $second) = ($second, $first);

This correctly interchanges the values of $first and $second, without the use of a temporary variable.

Assignment in Functional Programming Languages

All of the identifiers used in pure functional languages and some of them used in other functional languages are just names of values. As such, their values never change. For example, in ML, names are bound to values with the val declaration, whose form is exemplified in the following:

val cost = quantity * price;

If cost appears on the left side of a subsequent val declaration, that declaration creates a new version of the name cost, which has no relationship with the previous version, which is then hidden.

CONTROL STRUCTURES:

Selecting among alternative control flow paths (of statement execution) and some means of repeated execution of statements or sequences of statements. Statements that provide these kinds of capabilities are called control statements.

SELECTION STATEMENTS:

A selection statement provides the means of choosing between two or more execution paths in a program. Selection statements fall into two general categories: two-way and n-way, or multiple selection.

Two-Way Selection Statements:

The general form of a two-way selector is as follows:

```
if control_expression
then clause
else clause
```

The Control Expression:

Downloaded From EnggTree.com

EnggTree.com

Control expressions are specified in parentheses if the then reserved word is not used to introduce the then clause. In those cases where the then reserved word is used, there is less need for the parentheses, so they are often omitted, as in Ruby.

Clause Form:

In many languages, the then and else clauses appear as either single statements or compound statements. Many languages use braces to form compound statements, which serve as the bodies of then and else clauses. In Python and Ruby, the then and else clauses are statement sequences, rather than compound statements. The complete selection statement is terminated in these languages with the reserved word.

For example,

```
if x > y :
x = y
print "case 1"
```

Notice that rather than then, a colon is used to introduce the then clause in Python.

Nesting Selectors:

That ambiguous grammar was as follows:

<if_stmt>→ if <expression>then <statement>

| if <expression> then <statement> else <statement>

Consider the following Java-like code:

```
if (sum == 0)
if (count == 0)
result = 0;
else
result = 1;
```

This statement can be interpreted in two different ways, depending on whether the else clause is matched with the first then clause or the second. Notice that the indentation seems to indicate that the else clause belongs with the first then clause.

The crux of the problem in this example is that the else clause follows two then clauses with no intervening else clause, and there is no syntactic indicator to specify a matching of the else clause to one of the then clauses. In Java, as in that the else clause is always paired with the nearest previous unpaired then clause. So, in the example, the else clause would be paired with the second then clause.

To force the alternative semantics in Java, the inner if is put in a compound, as in

```
if (sum == 0)
{
 if (count == 0)
 result = 0;
}
```

Downloaded From EnggTree.com

```
else
result = 1;
```

Perl requires that all then and else clauses be compound, it does not. In Perl, the previous code would be written as follows:

```
if (sum == 0)
{
if (count == 0)
{ result = 0; }
}
else
{
 result = 1;
 }
```

If the alternative semantics were needed, it would be

```
if (sum == 0)
{ if (count == 0)
{ result = 0; }
else
{ result = 1; }
}
```

Another way to avoid the issue of nested selection statements is to use an alternative means of forming compound statements. Consider the syntactic structure of the Java if statement. The then clause follows the control expression and the else clause is introduced by the reserved word else. When the then clause is a single statement and the else clause is present, although there is no need to mark the end, the else reserved word in fact marks the end of the then clause. When the then clause is a compound, it is terminated by a right brace. However, if the last clause in an if, whether then or else, is not a compound, there is no syntactic entity to mark the end of the whole selection statement.

consider the following Ruby statement:

```
if a > b then sum = sum + a
        acount = acount + 1
else sum = sum + b
        bcount = bcount + 1
end
```

The design of this statement is more regular than that of the selection statements of the C-based languages, because the form is the same regardless of the number of statements in the then and else clauses. The first interpretation of the selector example at the beginning of this section, in which the else clause is matched to the nested if, can be written in Ruby as follows:

```
if sum == 0 then
        if count == 0 then
           result = 0
        else result = 1
        end
end
```

Downloaded From EnggTree.com

EnggTree.com

Because the end reserved word closes the nested if, it is clear that the else clause is matched to the inner then clause.

The second interpretation of the selection statement at the beginning of this section, in which the else clause is matched to the outer if, can be written in Ruby as follows:

```
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
    result = 1
end
```

The following statement, written in Python, is semantically equivalent to the last Ruby statement above:

```
if sum == 0 :
  if count == 0 :
     result = 0
else:
    result = 1
```

Selector Expressions:

Consider the following example selector written in F#:

```
let y = if x > 0 then x
        else 2 * x;;
```

This creates the name y and sets it to either x or 2 * x, depending on whether x is greater than zero.

Multiple-Selection Statements:

The multiple-selection statement allows the selection of one of any number of statements or statement groups. It is, therefore, a generalization of a selector. In fact, two-way selectors can be built with a multiple selector. The need to choose from among more than two control paths in programs is common. Although a multiple selector can be built from two-way selectors and gotos.

Examples of Multiple Selectors:

The C multiple-selector statement, switch, which is also part of C++, Java, and JavaScript, is a relatively primitive design. Its general form is

switch (expression) {
 case constant_expression$_1$:statement$_1$;
. . .
 case constant$_n$: statement_$_n$;
[default: statement$_{n+1}$] }

The optional default segment is for unrepresented values of the control expression. If the value of the control expression is not represented and no default segment is present, then the statement does nothing.

Downloaded From EnggTree.com

EnggTree.com

The break statement, which is actually a restricted goto, is normally used for exiting switch statements. break transfers control to the first statement after the compound statement in which it appears.

The C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels.

```
switch (x)
        default:
        if (prime(x))
           case 2: case 3: case 5: case 7:
           process_prime(x);
         else
            case 4: case 6: case 8: case 9: case 10:
            process_composite(x);

switch (value) {
        case -1:
             Negatives++;
             break;
        case 0:
             Zeros++;
             goto case 1;
        case 1:
              Positives++;
        default:
            Console.WriteLine("Error in switch \n");
}
```

PHP's switch uses the syntax of C's switch but allows more type flexibility. The case values can be any of the PHP scalar types— string, integer, or double precision.

Ruby has two forms of multiple- selection constructs, both of which are called case expressions.

```
case
when Boolean_expression then expression
. . .
when Boolean_expression then expression
[else expression]
End

leap = case
 when year % 400 == 0 then true
 when year % 100 == 0 then false
 else year % 4 == 0
 end
```

Perl, Python, and Lua do not have multiple-selection statements.

Implementing Multiple Selection Structures:

Downloaded From EnggTree.com

EnggTree.com

A multiple selection statement is essentially an n-way branch to segments of code, where n is the number of selectable segments.

Consider again the general form of the C switch statement, with breaks:

```
switch (expression) {
     case constant_expression1: statement1;
     break;
      . . .
     case constantn: statementn;
      break;
      [default: statementn+1]
}
```

One simple translation of this statement follows:

Code to evaluate expression into t

```
goto branches
label1: code for statement1
         goto out
. . .
labeln: code for statementn
        goto out
default: code for statementn+1
        goto out
branches: if t = constant_expression1 goto label1
        . . .
        if t = constant_expressionn goto labeln
        goto default
out
```

ITERATIVE STATEMENT:

An iterative statement is one that causes a statement or collection of statements to be executed zero, one, or more times. An iterative statement is often called a loop.

The body of an iterative statement is the collection of statements whose execution is controlled by the iteration statement. We use the term pretest to mean that the test for loop completion occurs before the loop body is executed and posttest to mean that it occurs after the loop body is executed. The iteration statement and the associated loop body together form an iteration statement.

Counter-Controlled Loops:

A counting iterative control statement has a variable, called the loop variable, in which the count value is maintained. It also includes some means of specifying the initial and terminal values of the loop variable, and the difference between sequential loop variable values, often called the stepsize. The initial, terminal, and stepsize specifications of a loop are called the loop parameters.

The for Statement of the C-Based Languages:

The general form of C's for statement is

```
for (expression_1; expression_2; expression_3)
loop body
```

Downloaded From EnggTree.com

EnggTree.com

The loop body can be a single statement, a compound statement, or a null statement. The expressions in a for statement are often assignment statements. The first expression is for initialization and is evaluated only once, when the for statement execution begins. The second expression is the loop control and is evaluated before each execution of the loop body. The last expression in the for is executed after each execution of the loop body. It is often used to increment the loop counter.

```
expression_1
loop:
 if expression_2 = 0 goto out
 [loop body]
 expression_3
goto  loop
out: . . .
```

Following is an example of a skeletal C for statement:

```
for (count = 1; count <= 10; count++)
 . . .
}
```

The for Statement of Python:

The general form of Python's for is

```
for loop_variable in object:
 - loop body
else:
 - else clause
```

The loop variable is assigned the value in the object, which is often a range, one for each execution of the loop body. The else clause, when present, is executed if the loop terminates normally.

Consider the following example:

```
for count in [2, 4, 6]:
 print count
```

produces

```
2
4
6
```

Counter-Controlled Loops:

The general form of an F# function for simulating counting loops, named forLoop in this case, is as follows:

```
let rec forLoop loopBody reps =
 if reps <= 0 then
 ()
 else
 loopBody()
 forLoop loopBody, (reps - 1);;
```

Downloaded From EnggTree.com

EnggTree.com

In this function, the parameter loopBody is the function with the body of the loop and the parameter reps is the number of repetitions. The reserved word rec appears before the name of the function to indicate that it is recursive.

Logically Controlled Loops

In many cases, collections of statements must be repeatedly executed, but the repetition control is based on the Boolean expression rather than counter.

The pretest and posttest logical loops have the following forms:

```
while (control_expression)
 loop body
```

and

```
do
 loop body
while (control_expression);
```

The operational semantics descriptions of those two statements follow:

```
while
loop:
 if control_expression is false goto out
 [loop body]
goto loop
out: . . .
```

```
do-while
loop:
 [loop body]
 if control_expression is true goto loop
```

User-Located Loop Control Mechanisms:

In some situations, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body. Such loops have the structure of infinite loops but include one or more user-located loop exits.

C, C++, Python, Ruby, and C# have unconditional unlabelled exits (break). Java and Perl have unconditional labelled exits (break in Java, last in Perl).

Following is an example of nested loops in Java, in which there is a break out of the outer loop from the nested loop:

```
outerLoop:
 for (row = 0; row < numRows; row++)
 for (col = 0; col < numCols; col++) {
 sum += mat[row][col];
 if (sum > 1000.0)
 break outerLoop;
 }
```

Downloaded From EnggTree.com

EnggTree.com

continue, that transfers control to the control mechanism of the smallest enclosing loop. This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop construct. For example, consider the following:

```
while (sum < 1000) {
 getnext(value);
 if (value < 0) continue;
 sum += value;
```

A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop. On the other hand, in

```
while (sum < 1000) {
 getnext(value);
 if (value < 0) break;
 sum += value;}
```

a negative value terminates the loop.

UNCONDITIONAL BRANCHING:

An unconditional branch statement transfers execution control to a specified location in the program. The unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements.

Without usage restrictions, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain.

Restricting gotos so they can transfer control only downward in a program partially alleviates the problem. It allows gotos to transfer control around code sections in response to errors or unusual conditions but disallows their use to build any sort of loop.

A few languages have been designed without a goto— for example, Java, Python, and Ruby.

GUARDED COMMANDS:

Dijkstra's selection statement has the form

```
if <Boolean expression> -> <statement>
[] <Boolean expression> -> <statement>
[] . . .
[] <Boolean expression> -> <statement>
fi
```

The closing reserved word, fi, is the opening reserved word spelled backward. The small blocks, called fatbars, are used to separate the guarded clauses and allow the clauses to be statement sequences. Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a guarded command.

This selection statement has the appearance of a multiple selection, but its semantics is different. All of the Boolean expressions are evaluated each time the statement is reached during execution. If more than one expression is true, one of the corresponding statements can be nondeterministically chosen for execution. An implementation might always choose the statement associated with the first Boolean expression that evaluates to be true.

Consider the following
example:

```
if i = 0 -> sum := sum + i
```

Downloaded From EnggTree.com

EnggTree.com

```
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

If i = 0 and j > i, this statement chooses nondeterministically between the first and third assignment statements. If i is equal to j and is not zero, a runtime error occurs because none of the conditions are true.

For example, to find the largest of two numbers, we can use

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

This computes the desired result without overspecifying the solution. In particular, if x and y are equal, it does not matter which we assign to max. This is a form of abstraction provided by the nondeterministic semantics of the statement.

Now, consider this same process coded in a traditional programming language selector:

```
if (x >= y)
 max = x;
else
 max = y;
```

This could also be coded as follows:

```
if (x > y)
max = x;
else
 max = y;
```

There is no practical difference between these two statements. The first assigns x to max when x and y are equal; the second assigns y to max in the same circumstance.

The loop structure proposed by Dijkstra has the form

```
do <Boolean expression> -> <statement>
[] <Boolean expression> -> <statement>
[] . . .
[] <Boolean expression> -> <statement>
od
```

The semantics of this statement is that all Boolean expressions are evaluated on each iteration. If more than one is true, one of the associated statements is nondeterministically (perhaps randomly) chosen for execution, after which the expressions are again evaluated. When all expressions are simultaneously false, the loop terminates.

Program verification is virtually impossible when goto statements are used. Verification is greatly simplified if (1) only logical loops and selections are used or (2) only guarded commands are used.

There is increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts.

Downloaded From EnggTree.com

UNIT III

SUBPROGRAMS AND IMPLEMENTATIONS

FUNDAMENTALS OF SUBPROGRAM:

Subprograms are the fundamental building blocks of programs. A subprogram definition describes the interface to and the actions of the subprogram abstraction. A subprogram call is the explicit request that a specific subprogram be executed. A subprogram is said to be active if, after having been called, it has begun execution but has not yet completed that execution.

A subprogram header, which is the first part of the definition, serves several purposes. First, it specifies that the following syntactic unit is a subprogram definition. In languages that have more than one kind of subprogram, the kind of the subprogram is usually specified with a special word. Second, if the subprogram is not anonymous, the header provides a name for the subprogram. Third, it may specify a list of parameters.

Consider the following header examples:

def adder (parameters):

This is the header of a Python subprogram named adder. Ruby subprogram headers also begin with def. The header of a JavaScript subprogram begins with function. In C, the header of a function named adder might be as follows:

void adder (parameters)

One characteristic of Python functions that sets them apart from the functions of other common programming languages is that function def statements are executable. When a def statement is executed, it assigns the given name to the given function body. Until a function's def has been executed, the function cannot be called. Consider the following skeletal example:

```
if . . .
 def fun(. . .):
 . . .
else
 def fun(. . .):
 . . .
```

If the then clause of this selection construct is executed, that version of the function fun can be called, but not the version in the else clause. Likewise, if the else clause is chosen, its version of the function can be called but the one in the then clause cannot.

Ruby methods are often defined in class definitions but can also be defined outside class definitions, in which case they are considered methods of the root object. Such methods can be called without an object receiver, as if they were functions in C or C++. If a Ruby method is

Downloaded From EnggTree.com

EnggTree.com

called without a receiver, self is assumed. If there is no method by that name in the class, enclosing classes are searched, up to Object, if necessary.

Parameters:

There are two ways that a non-method subprogram can gain access to the data that it is to process: through direct access to nonlocal variables (declared elsewhere but visible in the subprogram) or through parameter passing. Data passed through parameters are accessed using names that are local to the subprogram.

The parameters in the subprogram header are called formal parameters. They are sometimes thought of as dummy variables because they are not variables in the usual sense: In most cases, they are bound to storage only when the subprogram is called, and that binding is often through some other program variables.

Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called actual parameters.

In most programming languages, the correspondence between actual and formal parameters—or the binding of actual parameters to formal parameters— is done by position: The first actual parameter is bound to the first formal parameter and so forth. Such parameters are called positional parameters.

When parameter lists are long, however, it is easy for a programmer to make mistakes in the order of actual parameters in the list. One solution to this problem is to provide keyword parameters, in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call. The advantage of keyword parameters is that they can appear in any order in the actual parameter list. Python functions can be called using this technique, as in,

```
sumer(length = my_length,
 list = my_array,
 sum = my_sum)
```

where the definition of sumer has the formal parameters length, list, and sum.

The disadvantage to keyword parameters is that the user of the subprogram must know the names of formal parameters.

In Python, Ruby, C++, and PHP, formal parameters can have default values. A default value is used if no actual parameter is passed to the formal parameter in the subprogram header.

```
def compute_pay(income, exemptions = 1, tax_rate)
```

For example, consider the following call:

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

A C++ function header for the compute_pay function can be written as follows:

```
float compute_pay(float income, float tax_rate, int exemptions = 1)
```

Downloaded From EnggTree.com

Notice that the parameters are rearranged so that the one with the default value is last. An example call to the C++ compute_pay function is
pay = compute_pay(20000.0, 0.15);

Procedures and Functions:

There are two distinct categories of subprograms—procedures and functions— both of which can be viewed as approaches to extending the language. Subprograms are collections of statements that define parameterized computations. Functions return values and procedures do not. In most languages that do not include procedures as a separate form of subprogram, functions can be defined not to return values and they can be used as procedures.

Procedures can produce results in the calling program unit by two methods: (1) If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them; and (2) if the procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed.
Functions structurally resemble procedures but are semantically modeled on mathematical functions.

If a function is a faithful model, it produces no side effects; that is, it modifies neither its parameters nor any variables defined outside the function.

Functions are called by appearances of their names in expressions, along with the required actual parameters. The value produced by a function's execution is returned to the calling code, effectively replacing the call itself.

In C++,
float power(float base, float exp)
which could be called with
result = 3.4 * power(10.0, x)

The standard C++ library already includes a similar function named pow. Compare this with the same operation in Perl, in which exponentiation is a built-in operation:
result = 3.4 * 10.0 ** x

DESIGN ISSUES FOR SUBPROGRAMS:

Subprograms are complex structures, and it follows from this that a lengthy list of issues is involved in their design.

One obvious issue is the choice of one or more parameter- passing methods that will be used.

A closely related issue is whether the types of actual parameters will be type checked against the types of the corresponding formal parameters.

The most important question here is whether local variables are statically or dynamically allocated.

Next, there is the question of whether subprogram definitions can be nested.

Another issue is whether subprogram names can be passed as parameters.

Downloaded From EnggTree.com

side effects of functions can cause problems.

The types and number of values that can be returned from functions are other design issues.

The question of whether subprograms can be overloaded or generic.

An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment. A generic subprogram is one whose computation can be done on data of different types in different calls. A closure is a nested subprogram which allow the subprogram to be called from anywhere in a program.

LOCAL REFERENCING ENVIRONMENTS:

Variables that are defined inside subprograms are called local variables, because their scope is usually the body of the subprogram in which they are defined. Local variables can be either static or stack dynamic. If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminates.

Advantages of stack- dynamic local variable:

flexibility.

The storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms.

Disadvantages:

The cost of the time required to allocate, initialize , and deallocate such variables.

Subprograms cannot be history sensitive; that is, they cannot retain data values of local variables between calls.

Advantages of static local variable:

The advantage of static local variables is that they are slightly more efficient—they require no run-time over - head for allocation and deallocation.

Disadvantage of static local variables:

The greatest disadvantage of static local variables is their inability to support recursion.

In C and C++ functions, locals are stack dynamic unless specifically declared to be static.

For example, in the following C (or C++) function, the variable sum is static and count is stack dynamic.

```
int adder(int list[], int listlen) {
 static int sum = 0;
 int count;
 for (count = 0; count < listlen; count ++)
 sum += list [count];
 return sum;
}
```

The methods of C++, Java, and C# have only stack-dynamic local variables.

Downloaded From EnggTree.com

In Python, Any variable declared to be global in a method must be a variable defined outside the method. A variable defined outside the method can be referenced in the method without declaring it to be global, but such a variable cannot be assigned in the method. If the name of a global variable is assigned in a method, it is implicitly declared to be a local and the assignment does not disturb the global. All local variables in Python methods are stack dynamic.

Nested Subprograms:

If a subprogram is needed only within another subprogram then nested subprogram is used.All the descendants of C, do not allow subprogram nesting. JavaScript, Python, Ruby, and Lua allow subprograms to be nested.

PARAMETER PASSING METHODS:

Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms.

Semantics Models of Parameter Passing:

Formal parameters are characterized by one of three distinct semantics models:

(1) They can receive data from the corresponding actual parameter;

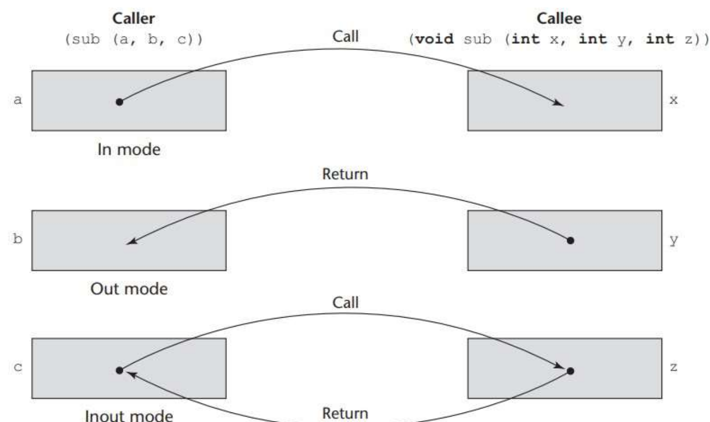
(2) they can transmit data to the actual parameter; or

(3) they can do both. These models are called in mode, out mode, and inout mode, respectively.

For example, consider a subprogram that takes two arrays of int values as parameters—list1 and list2. The subprogram must add list1 to list2 and return the result as a revised version of list2. Furthermore, the subprogram must create a new array from the two given arrays and return it. For this subprogram, list1 should be in mode, because it is not to be changed by the subprogram. list2 must be inout mode, because the subprogram needs the given value of the array and must return its new value. The third array should be out mode, because there is no initial value for this array and its computed value must be returned to the caller.

Figure 9.1

The three semantics models of parameter passing when physical moves are used



Downloaded From EnggTree.com

Implementation Models of Parameter Passing:

A variety of models have been developed by language designers to guide the implementation of the three basic parameter transmission modes.

Pass-by-Value:

When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, thus implementing in-mode semantics.

Pass-by- value is normally implemented by copy, because accesses often are more efficient with this approach. It could be implemented by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell.

The advantage of pass-by-value is that for scalars it is fast, in both linkage cost and access time.

The main disadvantage of the pass-by-value method if copies are used is that additional storage is required for the formal parameter.

Pass-by-Result:

Pass-by-result is an implementation model for out-mode parameters. When a parameter is passed by result, no value is transmitted to the subprogram. The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the caller's actual parameter, which obviously must be a variable. Pass-by-result also requires the extra storage.

One problem with the pass-by-result model is that there can be an actual parameter collision, such as the one created with the call.

```
sub(p1, p1)
```

In sub, assuming the two formal parameters have different names, the two can obviously be assigned different values. Then, whichever of the two is copied to their corresponding actual parameter last becomes the value of p1 in the caller. Thus, the order in which the actual parameters are copied determines their value.

For example, consider the following C# method, which specifies the pass-by-result method with the out specifier on its formal parameter.

```
void Fixer(out int x, out int y) {
 x = 17;
 y = 35;
}
. . .
f.Fixer(out a, out a);
```

Pass-by-Value-Result:

Pass-by-value-result is an implementation model for inout-mode parameters in which actual values are copied. It is in effect a combination of pass-by-value and pass-by-result. The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable.

Downloaded From EnggTree.com

Pass-by-value-result is sometimes called pass-by-copy, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

Pass-by-Reference:

Pass-by-reference is a second implementation model for inout-mode parameters. The pass-by- reference method transmits an access path, usually just an address, to the called subprogram. The actual parameter is shared with the called subprogram.

The advantage of pass- by- reference is that the passing process itself is efficient, in terms of both time and space. Duplicate space is not required and no copying is required.

several disadvantages to the pass-by-reference method. First, access to the formal parameters will be slower than pass-by-value parameters, because of the additional level of indirect addressing that is required. Second, if only one-way communication to the called subprogram is required, erroneous changes may be made to the actual parameter. Another problem of pass-by-reference is that aliases can be created.

Consider a C++ function that has two parameters that are to be passed by reference, as in

void fun(int &first, int &second)

If the call to fun happens to pass the same variable twice, as in

fun(total, total)

then first and second in fun will be aliases.

Second, collisions between array elements can also cause aliases. For example, suppose the function fun is called with two array elements that are specified with variable subscripts, as in

fun(list[i], list[j])

If these two parameters are passed by reference and i happens to be equal to j, then first and second are again aliases.

Third, if two of the formal parameters of a subprogram are an element of an array and the whole array, and both are passed by reference, then a call such as

fun1(list[i], list)

could result in aliasing in fun1, because fun1 can access all elements of list through the second parameter and access a single element through its first parameter.

Pass-by-Name:

When parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. A passby-name formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced. Implementing a pass-by-name parameter requires a subprogram to be passed to the called subprogram to evaluate the address or value of the formal parameter. Pass-by-name parameters are both complex to implement and inefficient.

Downloaded From EnggTree.com

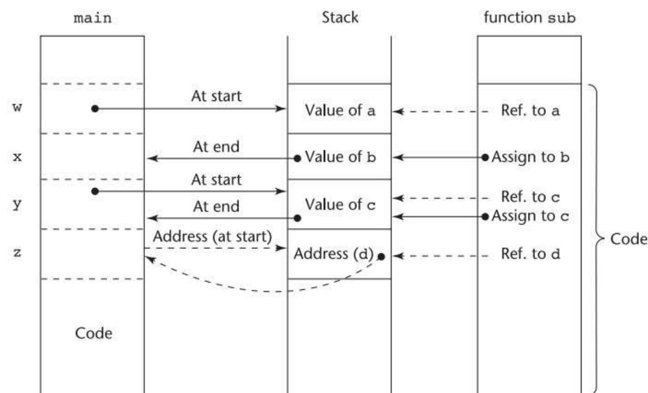
EnggTree.com

Implementing Parameter-Passing Methods:

Pass-by- value parameters have their values copied into stack locations. The stack locations then serve as storage for the corresponding formal parameters. Pass-by-result parameters are implemented as the opposite of pass- by- value. The values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram. Pass-by- value- result parameters can be implemented directly from their semantics as a combination of pass-by-value and pass-byresult. The stack location for such a parameter is initialized by the call and is then used like a local variable in the called subprogram. Pass- by- reference parameters are perhaps the simplest to implement. Regardless of the type of the actual parameter, only its address must be placed in the stack.

Figure 9.2

One possible stack implementation of the common parameter-passing methods



Function header: `void sub (int a, int b, int c, int d)`
Function call in `main`: `sub (w,x,y,z)`
(pass w by value, x by result, y by value-result, z by reference)

Activate \
Go to Settin

Subprogram sub is called from main with the call sub(w, x, y, z), where w is passed by value, x is passed by result, y is passed by value-result, and z is passed by reference.

Examples of Parameter Passing:
Consider the following C function:

```
void swap1(int a, int b) {
 int temp = a;
 a = b;
 b = temp;
}
```

Suppose this function is called with swap1(c, d);

Recall that C uses pass-by-value. The actions of swap1 can be described by the following pseudocode:

```
a = c        – Move first parameter value in
b = d        – Move second parameter value in
temp = a
a = b
```

Downloaded From EnggTree.com

EnggTree.com

b = temp

We can modify the C swap function to deal with pointer parameters to achieve the effect of pass-by-reference:

```
void swap2(int *a, int *b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}
```

swap2 can be called with

```
swap2(&c, &d);
```

The actions of swap2 can be described with the following:

```
a = &c – Move first parameter address in
b = &d – Move second parameter address in
temp = *a
*a = *b
*b = temp
```

C++ using reference parameters as follows:

```
void swap2(int &a, int &b) {
 int temp = a;
 a = b;
 b = temp;
}
```

OVERLOADED SUBPROGRAMS:

An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment. Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its parameters, and possibly in its return type. The meaning of a call to an overloaded subprogram is determined by the actual parameter list and/or possibly the type of the returned value.

Each version of an overloaded subprogram has a unique parameter profile, the compiler can disambiguate occurrences of calls to them by the different type parameters. Unfortunately, it is not that simple. Parameter coercions, when allowed, complicate the disambiguation process enormously. Simply stated, the issue is that if no method's parameter profile matches the number and types of the actual parameters in a method call, but two or more methods have parameter profiles that can be matched through coercions, which method should be called? the compiler can choose the method that "best" matches the call.

Downloaded From EnggTree.com

C++, Java, and C# allow mixed-mode expressions, the return type is irrelevant to disambiguation of overloaded functions.
For example, if a C++ program has two functions named fun and both take an int parameter but one returns an int and one returns a float, the program would not compile, because the compiler could not determine which version of fun should be used.

```
void fun(float b = 0.0);
void fun(int);
. . .
fun();
```

The call is ambiguous and will cause a compilation error.

GENERIC SUBPROGRAMS:

A polymorphic subprogram takes parameters of different types on different activations. Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism. Overloaded subprograms need not behave similarly. Languages that support object-oriented programming usually support sub - type polymorphism. Subtype polymorphism means that a variable of type T can access any object of type T or any type derived from T.
A more general kind of polymorphism is provided by the methods of Python and Ruby. Recall that variables in these languages do not have types, so formal parameters do not have types. Therefore, a method will work for any type of actual parameter, as long as the operators used on the formal parameters in the method are defined. Parametric polymorphism is provided by a subprogram that takes generic parameters that are used in type expressions that describe the types of the parameters of the subprogram.

Generic Functions in C++:

Generic functions in C++ have the descriptive name of template functions. The definition of a template function has the general form

```
template <template parameters>
—a function definition that may include the template parameters
```

A template parameter (there must be at least one) has one of the forms class identifier
typename identifier
The class form is used for type names. The typename form is used for passing a value to the template function. For example, it is sometimes convenient to pass an integer value for the size of an array in the template function.
As an example of a template function, consider the following:

```
template <class Type>
Type max(Type first, Type second) {
 return first > second ? first : second;
}
```

Downloaded From EnggTree.com

where Type is the parameter that specifies the type of data on which the function will operate. This template function can be instantiated for any type for which the operator > is defined. For example, if it were instantiated with int as the parameter, it would be

```
int max(int first, int second) {
 return first > second ? first : second;
}
```

Although this process could be defined as a macro, a macro would have the disadvantage of not operating correctly if the parameters were expressions with side effects. For example, suppose the macro were defined as

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
```

This definition is generic in the sense that it works for any numeric type. However, it does not always work correctly if called with a parameter that has a side effect, such as

```
max(x++, y)
```

which produces

```
((x++) > (y) ? (x++) : (y))
```

Whenever the value of x is greater than that of y, x will be incremented twice.

Generic Methods in Java 5.0:

The name of a generic class in Java 5.0 is specified by a name followed by one or more type variables delimited by pointed brackets. For example,

```
generic_class<T>
```

where T is the type variable.

Java's generic methods differ from the generic subprograms of C++ in several important ways. First, generic parameters must be classes—they cannot be primitive types. Second, although Java generic methods can be instantiated any number of times, only one copy of the code is built. Third, in Java, restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters. Such restrictions are called bounds.

consider the following skeletal method definition:

```
public static T doIt(T[] list) { . . . }
```

This defines a method named doIt that takes an array of elements of a generic type. The name of the generic type is T and it must be an array. Following is an example call to doIt:

```
doIt<String>(myList);
```

Now, consider the following version of doIt, which has a bound on its generic parameter:

```
public static <T extends comparable> T doIt(T[] list) { . . . }
```

The expression specifies that T should be a "subtype" of the bounding type. So, in this context, extends means the generic class (or interface) either extends the bounding class (the bound if it is a class) or implements the bounding interface.

Downloaded From EnggTree.com

Java 5.0 supports wildcard types. For example, Collection<?> is a wildcard type for collection classes.

```
void printCollection(Collection<?> c) {
 for (Object e: c) {
 System.out.println(e);
 }
}
```

This method prints the elements of any Collection class, regardless of the class of its components.

Generic Methods in C# 2005:

The generic methods of C# 2005 are similar in capability to those of Java 5.0, except there is no support for wildcard types.

```
class MyClass { public static T DoIt<T>(T p1) {
 . . .
} }
```

The method DoIt can be called without specifying the generic parameter if the compiler can infer the generic type from the actual parameter in the call. For example, both of the following calls are legal:

```
int myInt = MyClass.DoIt(17); // Calls DoIt<int>
string myStr = MyClass.DoIt('apples');
 // Calls DoIt<string>
```

Generic Functions in F#:

F# infers a generic type for the parameters and the return value. This is called automatic generalization. For example, consider the following function definition:

```
let getLast (a, b, c) = c;;
```

Because no type information was included, the types of the parameters and the return value are all inferred to be generic.

Functions can be defined to have generic parameters, as in the following example:

```
let printPair (x: 'a) (y: 'a) =
 printfn "%A %A" x y;;
```

The %A format specification is for any type. The apostrophe in front of the type named a specifies it to be a generic type.

DESIGN ISSUES FOR FUNCTIONS:

The following design issues are specific to functions:

- Are side effects allowed?

Downloaded From EnggTree.com

- What types of values can be returned?
- How many values can be returned?

Functional Side Effects:

Because of the problems of side effects of functions that are called in expressions parameters to functions should always be in-mode. This requirement effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals. Functions can have either pass-by-value or pass-by-reference parameters, thus allowing functions that cause side effects and aliasing.

Types of Returned Values:

C allows any type to be returned by its functions except arrays and functions. Both of these can be handled by pointer type return values. C++ is like C but also allows user-defined types, or classes, to be returned from its functions. Ada, Python, Ruby, and Lua are the only languages among current imperative languages whose functions (and/or methods) can return values of any type.

Number of Returned Values:

In most languages, only a single value can be returned from a function. Ruby allows the return of more than one value from a method. If a return statement in a Ruby method is not followed by an expression, nil is returned. If followed by one expression, the value of the expression is returned. If followed by more than one expression, an array of the values of all of the expressions is returned.

```
return 3, sum, index
```

If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

```
a, b, c = fun()
```

In F#, multiple values can be returned by placing them in a tuple and having the tuple be the last expression in the function.

SEMANTICS OF CALL AND RETURN:

The subprogram call and return operations are together called subprogram linkage. The implementation of subprograms must be based on the semantics of the subprogram linkage of the language being implemented.

A subprogram call in a typical language has numerous actions associated with it. The call process must include the implementation of whatever parameter- passing method is used.

If local variables are not static, the call process must allocate storage for the locals declared in the called subprogram and bind those variables to that storage. It must save the execution status of the calling program unit. The execution status is everything needed to resume

Downloaded From EnggTree.com

execution of the calling program unit. This includes register values, CPU status bits, and the environment pointer (EP).

The calling process also must arrange to transfer control to the code of the subprogram and ensure that control can return to the proper place when the subprogram execution is completed.

The required actions of a subprogram return are less complicated than those of a call. If the subprogram has parameters that are out mode or inout mode and are implemented by copy, the first action of the return process is to move the local values of the associated formal parameters to the actual parameters. Next, it must deallocate the storage used for local variables and restore the execution status of the calling program unit. Finally, control must be returned to the calling program unit.

IMPLEMENTING SIMPLE SUBPROGRAMS:

By “simple” we mean that subprograms cannot be nested and all local variables are static. The semantics of a call to a “simple” subprogram requires the following actions:

1. Save the execution status of the current program unit.
2. Compute and pass the parameters.
3. Pass the return address to the called.
4. Transfer control to the called.

The semantics of a return from a simple subprogram requires the following actions:

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. The execution status of the caller is restored.
4. Control is transferred back to the caller.

The call and return actions require storage for the following:

• Status information about the caller
• Parameters
• Return address
• Return value for functions
• Temporaries used by the code of the subprograms

These, along with the local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return control to the caller. The last three actions of a call clearly must be done by the caller. Saving the execution status of the caller could be done by either. In the case of the return, the first, third, and fourth actions must be done by the called. Once again, the restoration of the execution status of the caller could be done by either the caller or the called. The linkage actions of the called can occur at two different times, either at the beginning of its execution or at the end. These are sometimes called the prologue and

Downloaded From EnggTree.com

epilogue of the subprogram linkage. In the case of a simple subprogram, all of the linkage actions of the callee occur at the end of its execution, so there is no need for a prologue.

A simple subprogram consists of two separate parts: the actual code of the subprogram, which is constant, and the local variables and data listed .

The format, or layout, of the noncode part of a subprogram is called an activation record, because the data it describes are relevant only during the activation or execution of the subprogram.

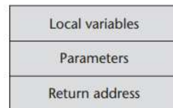


Figure 10.1
An activation record for simple subprogram

Figure 10.2 shows a program consisting of a main program and three subprograms: A, B, and C.

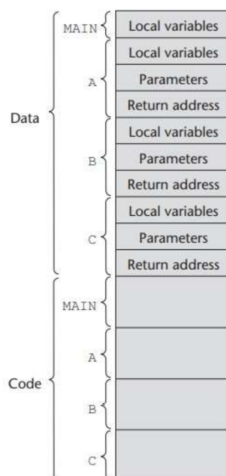


Figure 10.2
The code and activation records of a program with simple subprograms

The construction of the complete program shown in Figure 10.2 is not done entirely by the compiler. In fact, if the language allows independent compilation, the four program units—MAIN, A, B, and C—may have been compiled on different days, or even in different years. At the time each unit is compiled, the machine code for it, along with a list of references to external subprograms, is written to a file. The executable program shown in Figure 10.2 is put together by the linker, which is part of the operating system. When the linker is called for a main program, its first task is to find the files that contain the translated subprograms referenced in that program and load them into memory. Then, the linker must set the target addresses of all calls to those subprograms in the main program to the entry addresses of those subprograms. The same must be done for all calls to subprograms in the loaded subprograms and all calls to library subprograms.

IMPLEMENTING SUBPROGRAMS WITH STACK-DYANMIC LOCAL VARIABLES:

One of the most important advantages of stack- dynamic local variables is support for recursion.

More Complex Activation Records:

Downloaded From EnggTree.com

Subprogram linkage in languages that use stack- dynamic local variables are more complex than the linkage of simple subprograms for the following reasons:

• The compiler must generate code to cause the implicit allocation and dealocation of local variables.

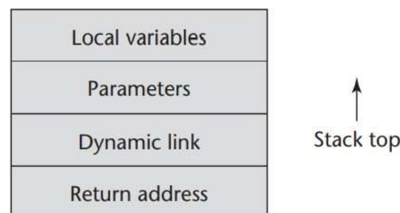
•Recursion adds the possibility of multiple simultaneous activations of a subprogram, which means that there can be more than one instance (incomplete execution) of a subprogram at a given time, with at least one call from outside the subprogram and one or more recursive calls. The number of activations is limited only by the memory size of the machine.

The format of an activation record for a given subprogram in most languages is known at compile time. In many cases, the size is also known for activation records because all local data are of a fixed size.

Some other languages, such as Ada, in which the size of a local array can depend on the value of an actual parameter. In those cases, the format is static, but the size can be dynamic. In languages with stack-dynamic local variables, activation record instances must be created dynamically.

Figure 10.3

A typical activation record for a language with stack-dynamic local variables



The dynamic link is a pointer to the base of the activation record instance of the caller. In staticscoped languages, this link is used to provide traceback information when a run-time error occurs. In dynamic-scoped languages, the dynamic link is used to access nonlocal variables.

Local scalar variables are bound to storage within an activation record instance. Local variables that are structures are sometimes allocated elsewhere, and only their descriptors and a pointer to that storage are part of the activation record.

Consider the following skeletal C function:

```
void sub(float total, int part) {
 int list[5];
 float sum;
 . . .
}
```

Activating a subprogram requires the dynamic creation of an instance of the activation record for the subprogram. As stated earlier, the format of the activation record is fixed at compile time, although its size may depend on the call in some languages. Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to create instances of these activation records on a stack. This stack is part of the runtime system and therefore is called the run-time stack, although we will usually just refer to it as the stack. Every subprogram activation, whether recursive or nonrecursive, creates a new instance of an activation

Downloaded From EnggTree.com

record on the stack. This provides the required separate copies of the parameters, local variables, and return address.

Figure 10.4

The activation record for function `sub`

Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

One more thing is required to control the execution of a subprogram— the EP. Initially, the EP points at the base, or first address of the activation record instance of the main program. The run-time system must ensure that it always points at the base of the activation record instance of the currently executing program unit. When a subprogram is called, the current EP is saved in the new activation record instance as the dynamic link. The EP is then set to point at the base of the new activation record instance. Upon return from the subprogram, the stack top is set to the value of the current EP minus one and the EP is set to the dynamic link from the activation record instance of the subprogram that has completed its execution. Resetting the stack top effectively removes the top activation record instance.

The EP is used as the base of the offset addressing of the data contents of the activation record instance—parameters and local variables. Note that the EP currently being used is not stored in the run-time stack. Only saved versions are stored in the activation record instances as the dynamic links.

The caller actions are as follows:

1. Create an activation record instance.
2. Save the execution status of the current program unit.
3. Compute and pass the parameters.
4. Pass the return address to the called.
5. Transfer control to the called.

The prologue actions of the called are as follows:

1. Save the old EP in the stack as the dynamic link and create the new value.
2. Allocate local variables.

The epilogue actions of the called are as follows:

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters.

2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.

Downloaded From EnggTree.com

EnggTree.com

3. Restore the stack pointer by setting it to the value of the current EP minus one and set the EP to the old dynamic link.

4. Restore the execution status of the caller.

5. Transfer control back to the caller

Parameters are not always transferred in the stack. In many compilers for RISC machines, parameters are passed in registers. This is because RISC machines normally have many more registers than CISC machines.

An Example Without Recursion:

Consider the following skeletal C program:

```
void fun1(float r) {
 int s, t;
 . . . <.............1
 fun2(s);
 . . .
}
void fun2(int x) {
 int y;
 . . . <.............2
 fun3(y);
 . .
}
void fun3(int q) {
 . . . <.............3
}
void main() {
 float p;
 . . .
 fun1(p);
 . . .
}
```

The sequence of function calls in this program is

main calls fun1

fun1 calls fun2

fun2 calls fun3

At point 1, only the activation record instances for function main and function fun1 are on the stack. When fun1 calls fun2, an instance of fun2’s activation record is created on the stack. When fun2 calls fun3, an instance of fun3’s activation record is created on the stack. When fun3’s execution ends, the instance of its activation record is removed from the stack, and the EP is used to reset the stack top pointer. Similar processes take place when functions fun2 and fun1

Downloaded From EnggTree.com

terminate. After the return from the call to fun1 from main, the stack has only the instance of the activation record of main.

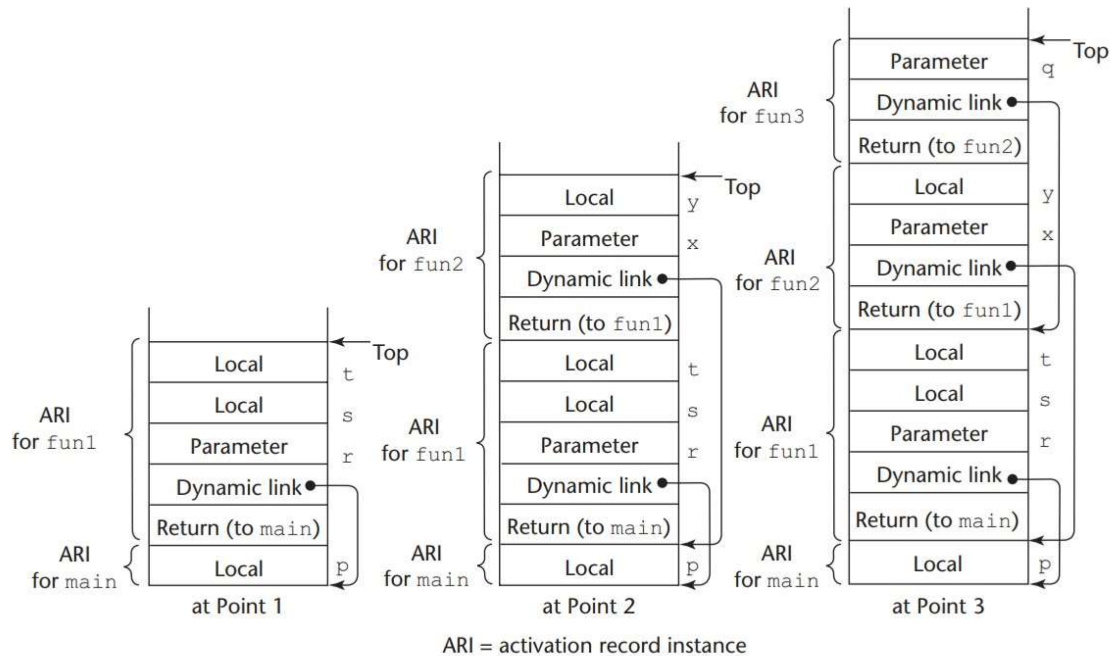


Figure 10.5

Stack contents for three points in a program

The collection of dynamic links present in the stack at a given time is called the dynamic chain, or call chain. It represents the dynamic history of how execution got to its current position, which is always in the subprogram code whose activation record instance is on top of the stack. References to local variables can be represented in the code as offsets from the beginning of the activation record of the local scope, whose address is stored in the EP. Such an offset is called a local_offset. The local_offset of a variable in an activation record can be determined at compile time, using the order, types, and sizes of variables declared in the subprogram associated with the activation record.

Recursion:

Consider the following example C program, which uses recursion to compute the factorial function:

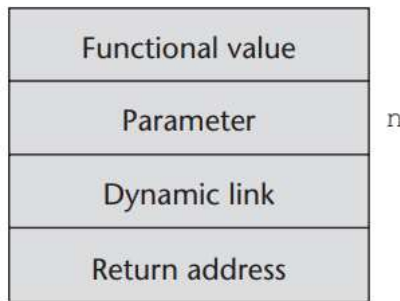
```
int factorial(int n) {
 <..............1
 if (n <= 1)
 return 1;
 else return (n * factorial(n - 1));
 <..............2
 }
void main() {
```

```
int value;
value = factorial(3);
<..............3
}
```

Figure 10.7 shows the contents of the stack for the three times execution reaches position 1 in the function factorial. Each shows one more activation of the function, with its functional value undefined. The first activation record instance has the return address to the calling function, main. The others have a return address to the function itself; these are for the recursive calls. Figure 10.8 shows the stack contents for the three times that execution reaches position 2 in the function factorial. Position 2 is meant to be the time after the return is executed but before the activation record has been removed from the stack. Recall that the code for the function multiplies the current value of the parameter n by the value returned by the recursive call to the function.

Figure 10.6

The activation record for `factorial`



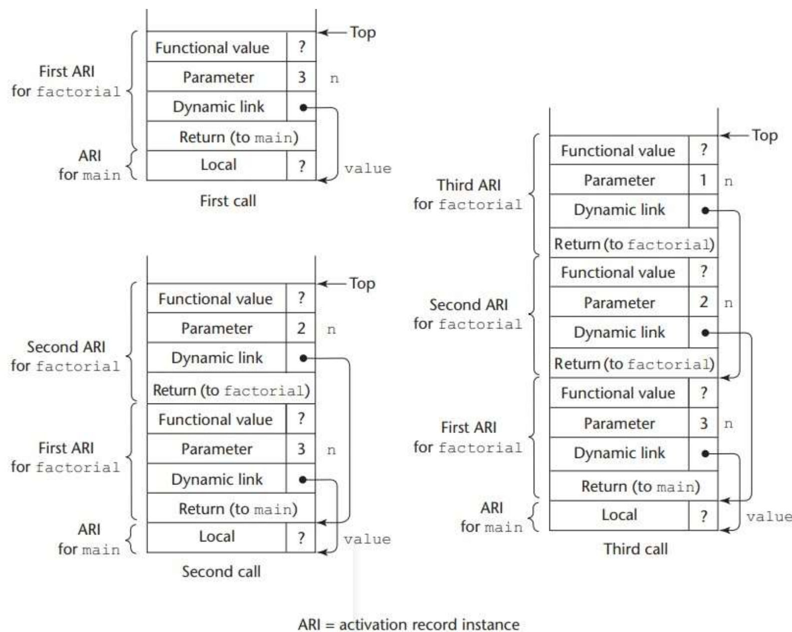


Figure 10.7

Stack contents at position 1 in `factorial`

Downloaded From EnggTree.com

The first return from factorial returns the value 1. The activation record instance for that activation has a value of 1 for its version of the parameter n. The result from that multiplication, 1, is returned to the second activation of factorial to be multiplied by its parameter value for n, which is 2. This step returns the value 2 to the first activation of factorial to be multiplied by its parameter value for n, which is 3, yielding the final functional value of 6, which is then returned to the first call to factorial in main.

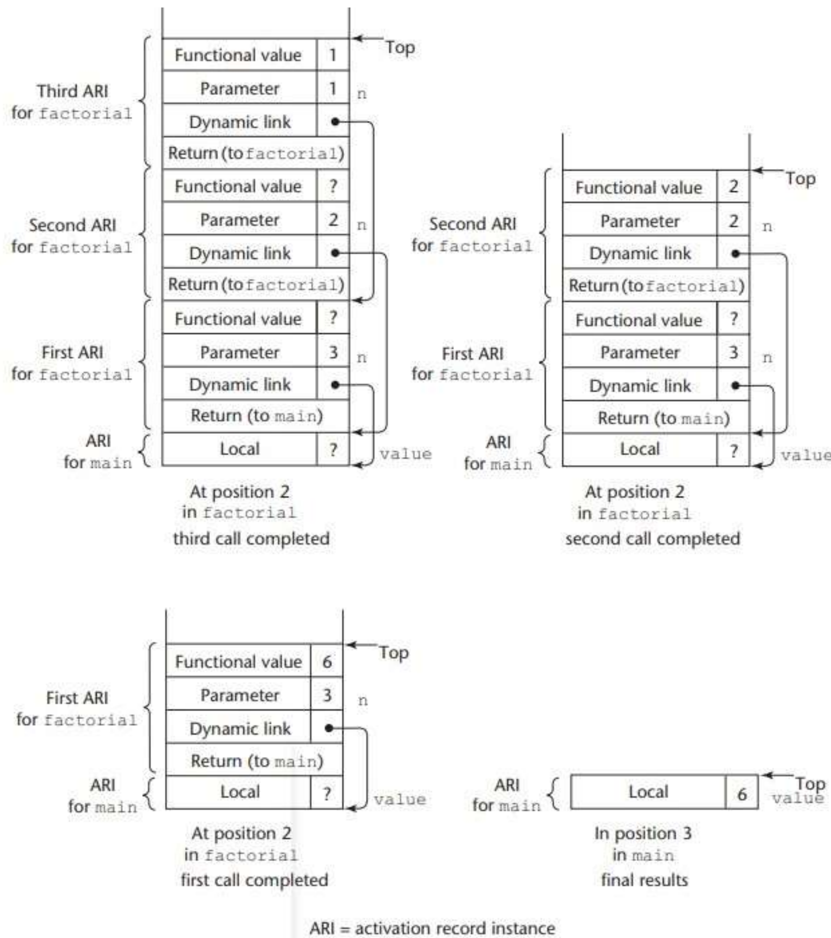


Figure 10.8

Stack contents during execution of main and factorial

NESTED SUBPROGRAMS:

The Basics:

The first step of the access process is to find the instance of the activation record in the stack in which the variable was allocated. The second part is to use the local_offset of the variable (within the activation record instance) to access it.

Finding the correct activation record instance is the more interesting and more difficult of the two steps. First, note that in a given subprogram, only variables that are declared in static ancestor scopes are visible and can be accessed. Also, activation record instances of all of the static ancestors are always on the stack when variables in them are referenced by a nested

Downloaded From EnggTree.com

EnggTree.com

subprogram. This is guaranteed by the static semantic rules of the static-scoped languages: A subprogram is callable only when all of its static ancestor subprograms are active.

Static Chains:

The most common way to implement static scoping in languages that allow nested subprograms is static chaining. In this approach, a new pointer, called a static link, is added to the activation record. The static link, which is sometimes called a static scope pointer, points to the bottom of the activation record instance of an activation of the static parent. It is used for accesses to nonlocal variables. Typically, the static link appears in the activation record below the parameters. The addition of the static link to the activation record requires that local offsets differ from when the static link is not included. Instead of having two activation record elements before the parameters, there are now three: the return address, the static link, and the dynamic link.

A static chain is a chain of static links that connect certain activation record instances in the stack. This chain can obviously be used to implement the accesses to nonlocal variables in static-scoped languages.

Finding the correct activation record instance of a nonlocal variable using static links is relatively straightforward. When a reference is made to a nonlocal variable, the activation record instance containing the variable can be found by searching the static chain until a static ancestor activation record instance is found that contains the variable.

Let static_depth be an integer associated with a static scope that indicates how deeply it is nested in the outermost scope. A program unit that is not nested inside any other unit has a static_depth of 0.

The length of the static chain needed to reach the correct activation record instance for a nonlocal reference to a variable X is exactly the difference between the static_depth of the subprogram containing the reference to X and the static_depth of the subprogram containing the declaration for X. This difference is called the nesting_depth, or chain_offset, of the reference.

```
# Global scope
 . . .
def f1():
def f2():
def f3():
. . . # end of f3
. . . # end of f2
 . . # end of f1
```

The static_depths of the global scope, f1, f2, and f3 are 0, 1, 2, and 3, respectively. If procedure f3 references a variable declared in f1, the chain_offset of that reference would be 2 (static_depth of f3 minus the static_depth of f1). If procedure f3 references a variable declared in f2, the chain_offset of that reference would be 1.

Downloaded From EnggTree.com

```
function main(){
 var x;
 function bigsub() {
 var a, b, c;
 function sub1  {
 var a, d;
 ...
 a = b + c; <.................................1
 ...
 } // end of sub1
 function sub2(x) {
 var b, e;
 function sub3() {
 var c, e;
 ...
 sub1();
 ...
 e = b + a; <................................2
 } // end of sub3
 ...
 sub3();
 ...
 a = d + e; <.................................3
 } // end of sub2
 ...
 sub2(7);
 ...
 } // end of bigsub
 ...
 bigsub();
 ...
} // end of main
```

The sequence of procedure calls is
main calls bigsub
bigsub calls sub2
sub2 calls sub3
sub3 calls sub1

At position 1 in procedure sub1, the reference is to the local variable, a, not to the nonlocal variable a from bigsub. This reference to a has the chain_offset/local_offset pair (0, 3). The reference to b is to the nonlocal b from bigsub. It can be represented by the pair (1, 4). The local_offset is 4, because a 3 offset would be the first local variable. if the dynamic link were used to do a simple search for an activation record instance with a declaration for the variable b, it would find the variable b declared in sub2, which would be incorrect. If the (1, 4) pair were used with the dynamic chain, the variable e from sub3 would be used. The static link, however,

Downloaded From EnggTree.com

points to the activation record for bigsub, which has the correct version of b. The variable b in sub2 is not in the referencing environment at this point and is (correctly) not accessible. The reference to c at point 1 is to the c defined in bigsub, which is represented by the pair (1, 5). After sub1 completes its execution, the activation record instance for sub1 is removed from the stack, and control returns to sub3. The reference to the variable e at position 2 in sub3 is local and uses the pair (0, 4) for access. The reference to the variable b is to the one declared in sub2, because that is the nearest static ancestor that contains such a declaration. It is accessed with the pair (1, 4).

The local_offset is 4 because b is the first variable declared in sub1, and sub2 has one parameter. The reference to the variable a is to the a declared in bigsub, because neither sub3 nor its static parent sub2 has a declaration for a variable amed a. It is referenced with the pair (2, 3).After sub3 completes its execution, the activation record instance for sub3 is removed from the stack, leaving only the activation record instances for main, bigsub, and sub2. At position 3 in sub2, the reference to the variablea is to the a in bigsub, which has the only declaration of a among the active routines. This access is made with the pair (1, 3). At this position, there is no visible scope containing a declaration for the variable d, so this reference to d is a static semantics error. The error would be detected when the compiler attempted to compute the chain_offset/local_offset pair. The reference to e is to the local e in sub2, which can be accessed with the pair (0, 5).

In summary, the references to the variable a at points 1, 2, and 3 would be represented by the following points:

- (0, 3) (local)
- (2, 3) (two levels away)
- (1, 3) (one level away)

It is reasonable at this point to ask how the static chain is maintained during program execution. If its maintenance is too complex, the fact that it is simple and effective would be unimportant. We assume here that parameters that are subprograms are not implemented.

Downloaded From EnggTree.com

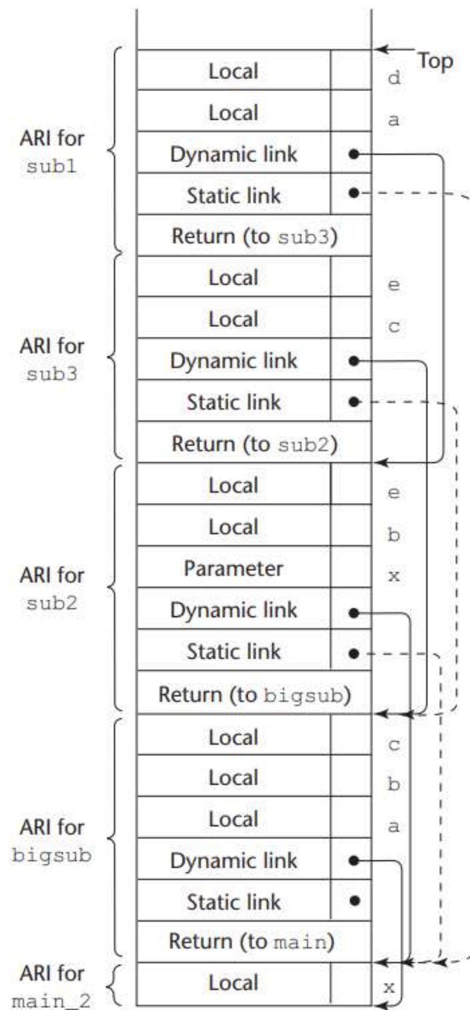


Figure 10.9

Stack contents at position 1 in the program main

The static chain must be modified for each subprogram call and return. The return part is trivial: When the subprogram terminates, its activation record instance is removed from the stack. After this removal, the new top activation record instance is that of the unit that called the subprogram whose execution just terminated. Because the static chain from this activation record instance was never changed, it works correctly just as it did before the call to the other subprogram. Therefore, no other action is required.

Consider again the program main and the stack situation shown in Figure 10.9. At the call to sub1 in sub3, the compiler determines the nesting_depth of sub3 (the caller) to be two levels inside the procedure that declared the called procedure sub1, which is bigsub. When the call to sub1 in sub3 is executed, this information is used to set the static link of the activation record instance for sub1. This static link is set to point to the activation record instance that is pointed to by the second static link in the static chain from the caller's activation record instance. In this case, the caller is sub3, whose static link points to its parent's activation record instance (that of sub2). The static link of the activation record instance for sub2 points to the activation record instance for bigsub. So, the static link for the new activation record instance for sub1 is set to

Downloaded From EnggTree.com

point to the activation record instance for bigsub. This method works for all subprogram linkage, except when parameters that are subprograms are involved. One criticism of using the static chain approach to access nonlocal variables is that references to variables in scopes beyond the static parent cost more than references to locals

BLOCKS:

C- based languages, provide for user-specified local scopes for variables called blocks. As an example of a block, consider the following code segment:

```
{ int temp;
 temp = list[upper];
 list[upper] = list[lower];
 list[lower] = temp;
}
```

A block is specified in the C-based languages as a compound statement that begins with one or more data definitions. The lifetime of the variable temp in the preceding block begins when control enters the block and ends when control exits the block. The advantage of using such a local is that it cannot interfere with any other variable with the same name that is declared elsewhere in the program, or more specifically, in the referencing environment of the block.

Blocks are treated as parameterless subprograms that are always called from the same place in the program. Therefore, every block has an activation record. An instance of its activation record is created every time the block is executed.

The maximum amount of storage required for block variables at any time during the execution of a program can be statically determined, because blocks are entered and exited in strictly textual order. This amount of space can be allocated after the local variables in the activation record.

For example, consider the following skeletal program:

```
void main() {
 int x, y, z;
 while ( . . ) {
 int a, b, c;
 .
 while ( . . . ) {
 int d, e;
 .
 }
 }
 while ( . . . ) {
 int f, g;
 . . .
}
```

Downloaded From EnggTree.com

EnggTree.com


```
 . .
}
```

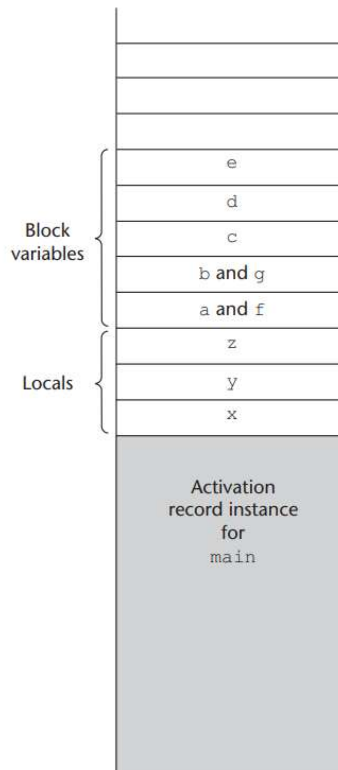


Figure 10.10

Block variable storage when blocks are not treated as parameterless procedures

IMPLEMENTING DYNAMIC SCOPING:

There are two distinct ways in which local variables and nonlocal references to them can be implemented in a dynamic-scoped language: deep access and shallow access.

DEEP ACCESS:

If local variables are stack dynamic and are part of the activation records in a dynamic-scoped language, references to nonlocal variables can be resolved by searching through the activation record instances of the other subprograms that are currently active, beginning with the one most recently activated. This concept is similar to that of accessing nonlocal variables in a static- scoped language with nested subprograms, except that the dynamic—rather than the static— chain is followed. The dynamic chain links together all subprogram activation record instances in the reverse of the order in which they were activated. Therefore, the dynamic chain is exactly what is needed to reference nonlocal variables in a dynamic-scoped language. This method is called deep access, because access may require searches deep into the stack.

Consider the following example skeletal program:

```
void sub3() {
 int x, z;
 x = u + v;
 . . .
}
```

Downloaded From EnggTree.com

EnggTree.com


```
void sub2() {
 int w, x;
 . . .
}
void sub1() {
 int v, w;
 . . .
}
void main() {
 int v, u;
 . . .
}
```

This program is written in a syntax that gives it the appearance of a program in a C- based language, but it is not meant to be in any particular language.

Suppose the following sequence of function calls occurs:

main calls sub1

sub1 calls sub2

sub2 calls sub3

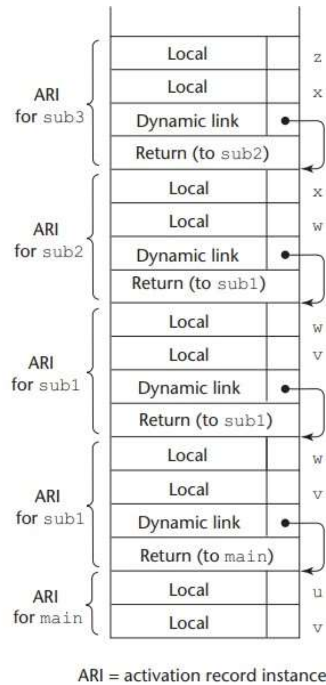
Consider the references to the variables x, u, and v in function sub3. The reference to x is found in the activation record instance for sub3. The reference to u is found by searching all of the activation record instances on the stack, because the only existing variable with that name is in main. This search involves following four dynamic links and examining 10 variable names. The reference to v is found in the most recent (nearest on the dynamic chain) activation record instance for the subprogram sub1.

Downloaded From EnggTree.com

EnggTree.com

Figure 10.11

Stack contents for a dynamic-scoped program



ARI = activation record instance

There are two important differences between the deep-access method for nonlocal access in a dynamic-scoped language and the static-chain method for static-scoped languages. First, in a dynamic-scoped language, there is no way to determine at compile time the length of the chain that must be searched. Second, activation records must store the names of variables for the search process, whereas in static-scoped language implementations only the values are required.

Shallow Access:

In the shallow-access method, variables declared in subprograms are not stored in the activation records of those subprograms. Because with dynamic scoping there is at most one visible version of a variable of any specific name at a given time, a very different approach can be taken.

One variation of shallow access is to have a separate stack for each variable name in a complete program. Every time a new variable with a particular name is created by a declaration at the beginning of a subprogram that has been called, the variable is given a cell at the top of the stack for its name. Every reference to the name is to the variable on top of the stack associated with that name, because the top one is the most recently created. When a subprogram terminates, the lifetimes of its local variables end, and the stacks for those variable names are popped. This method allows fast references to variables, but maintaining the stacks at the entrances and exits of subprograms is costly.

Another option for implementing shallow access is to use a central table that has a location for each different variable name in a program. Along with each entry, a bit called active is maintained that indicates whether the name has a current binding or variable association. Any access to any variable can then be to an offset into the central table. The offset is static, so the

Downloaded From EnggTree.com

EnggTree.com

access can be fast. SNOBOL implementations use the central table implementation technique. Maintenance of a central table is straightforward.

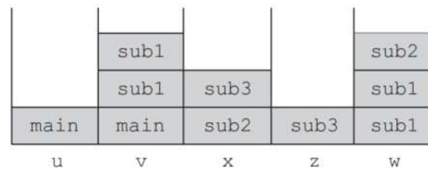
A subprogram call requires that all of its local variables be logically placed in the central table. If the position of the new variable in the central table is already active—that is, if it contains a variable whose lifetime has not yet ended (which is indicated by the active bit)—that value must be saved somewhere during the lifetime of the new variable. Whenever a variable begins its lifetime, the active bit in its central table position must be set.

There have been several variations in the design of the central table and in the way values are stored when they are temporarily replaced. One variation is to have a "hidden" stack on which all saved objects are stored. Because subprogram calls and returns, and thus the lifetimes of local variables, are nested, this works well.

The second variation is perhaps the cleanest and least expensive to implement. A central table of single cells is used, storing only the current version of each variable with a unique name. Replaced variables are stored in the activation record of the subprogram that created the replacement variable. This is a stack mechanism, but it uses the stack that already exists, so the new overhead is minimal. The choice between shallow and deep access to nonlocal variables depends on the relative frequencies of subprogram calls and nonlocal references. The deep access method provides fast subprogram linkage, but references to nonlocals, especially references to distant nonlocals (in terms of the call chain), are costly. The shallow-access method provides much faster references to nonlocals, especially distant nonlocals, but is more costly in terms of subprogram linkage.

Figure 10.12

One method of using shallow access to implement dynamic scoping



Downloaded From EnggTree.com

UNIT 4

OBJECT ORIENTATION, CONCURRENCY AND EVENT HANDLING

OBJECT ORIENTATION:

A language that is object oriented must provide support for three key language features: abstract data types, inheritance, and dynamic binding of method calls to methods.

Inheritance:

Abstract data types, with their encapsulation and access controls, are obvious candidates for reuse. The problem with the reuse of abstract data types is that, in nearly all cases, the features and capabilities of the existing type are not quite right for the new use. The old type requires at least some minor modifications. Such modifications can be difficult.

A second problem with programming with abstract data types is that the type definitions are all independent and are at the same level.

Inheritance offers a solution to both the modification problem posed by abstract data type reuse and the program organization problem. If a new abstract data type can inherit the data and functionality of some existing type, and is also allowed to modify some of those entities and add new entities, reuse is greatly facilitated without requiring changes to the reused abstract data type. Inheritance provides a framework for the definition of hierarchies of related classes that can reflect the descendant relationships in the problem space.

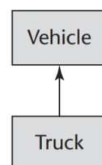
The abstract data types in object-oriented languages, are usually called classes. Class instances are called objects. A class that is defined through inheritance from another class is a derived class, a subclass, or a child class. A class from which the new class is derived is its base class, superclass, or parent class. The subprograms that define the operations on objects of a class are called methods. The calls to methods are sometimes called messages. The entire collection of methods of a class is called the message protocol, or message interface, of the class. Computations in an object-oriented program are specified by messages sent from objects to other objects, or in some cases, to classes.

Methods are similar to subprograms. Both are collections of code that perform some computation. Both can take parameters and return results. Passing a message is different from calling a subprogram. A subprogram typically processes data that is either passed to it by its caller as a parameter or is accessed nonlocally or globally. A message that is sent to an object is a request to execute one of its methods.

Figure 12.1 shows a simple diagram to indicate the relationship between the Vehicle class and the Truck class, in which the arrow points to the parent class.

Figure 12.1

A simple example of inheritance



There are several ways a derived class can differ from its parent.

1. The subclass can add variables and/or methods to those inherited from the parent class.

2. The subclass can modify the behavior of one or more of its inherited methods. A modified method has the same name, and often the same protocol, as the one of which it is a modification.

3. The parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass.

The new method is said to override the inherited method, which is then called an overridden method. The purpose of an overriding method is to provide an operation in the subclass that is similar to one in the parent class.

Classes can have two kinds of methods and two kinds of variables. The most commonly used methods and variables are called instance methods and instance variables. Every object of a class has its own set of instance variables, which store the object's state. The only difference between two objects of the same class is the state of their instance variables. For example, a class for cars might have instance variables for color, make, model, and year. Instance methods operate only on the objects of the class. Class variables belong to the class, rather than its object, so there is only one copy for the class.

If a new class is a subclass of a single parent class, then the derivation process is called single inheritance. If a class has more than one parent class, the process is called multiple inheritance.

One disadvantage of inheritance as a means of increasing the possibility of reuse is that it creates dependencies among the classes in an inheritance hierarchy.

Private members are visible inside the class, while public members also are visible to clients of the class. Private members of base class are not visible to subclasses, but public members are.

Dynamic Binding:

The third essential characteristic (after abstract data types and inheritance) of object-oriented programming languages is a kind of polymorphism provided by the dynamic binding of messages to method definitions. This is sometimes called dynamic dispatch. Consider the following situation: There is a base class, A, that defines a method draw that draws some figure associated with the base class. A second class, B, is defined as a subclass of A. Objects of this new class also need a draw method that is like that provided by A but a bit different because the subclass objects are slightly different. So, the subclass overrides the inherited draw method. If a client of A and B has a variable that is a reference to class A's objects, that reference also could point at class B's objects, making it a polymorphic reference. If the method draw, which is defined in both classes, is called through the polymorphic reference, the run-time system must determine, during execution, which method should be called, A's or B's

Downloaded From EnggTree.com

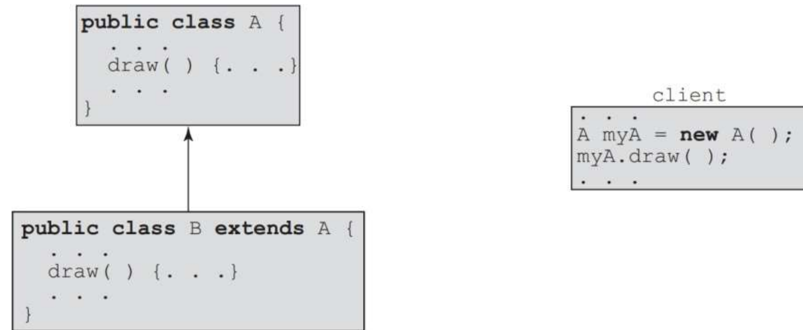


Figure 12.2
Dynamic binding

One purpose of dynamic binding is to allow software systems to be more easily extended during both development and maintenance.

In some cases, the design of an inheritance hierarchy results in one or more classes that are so high in the hierarchy that an instantiation of them would not make sense. For example, suppose a program defined a Building class and a collection of subclasses for specific types of buildings, for instance, French_Gothic_Cathedrals. It probably would not make sense to have an implemented draw method in Building. But because all of its descendant classes should have such methods, the protocol (but not the body) of that method is included in Building. Such a method is often called an abstract method (pure virtual method in C++). A class that includes at least one abstract method is called an abstract class (abstract base class in C++). Such a class usually cannot be instantiated, because some of its methods are declared but are not defined (they do not have bodies). Any subclass of an abstract class that is to be instantiated must provide implementations (definitions) of all of the inherited abstract methods.

DESIGN ISSUES FOR OBJECT ORIENTED LANGUAGES:

A number of issues must be considered when designing the programming language features to support inheritance and dynamic binding.

The Exclusivity of Objects:

Everything, from a simple scalar integer to a complete software system, is an object in this mind-set. The advantage of this choice is the elegance and pure uniformity of the language and its use. The primary disadvantage is that simple operations must be done through the message-passing process, which often makes them slower than similar operations in an imperative model, where single machine instructions implement such simple operations.

One alternative to the exclusive use of objects that is common in imperative languages to which support for object-oriented programming has been added is the following: Retain the complete collection of types from the base imperative language and add the object typing model.

Another alternative to the exclusive use of objects is to have an imperative style type structure for the primitive scalar types, but implement all structured types as objects.

Downloaded From EnggTree.com

Are Subclasses Subtypes?

If a language allows programs in which a variable of a class can be substituted for a variable of one of its ancestor classes in any situation, without causing type errors and without changing the behavior of the program, that language supports the principle of substitution.

In such a language, if class B is derived from class A, then B has everything A has and the behavior of an object of class B, when used in place of an object of class A, is identical to that of an object of class A. When this is true, B is a subtype of A.

The subtypes of Ada are examples of predefined subtypes.

For example,

subtype Small_Int is Integer range -100..100;

Variables of Small_Int type have all of the operations of Integer variables but can store only a subset of the values possible in Integer. Furthermore, every Small_Int variable can be used anywhere an Integer variable can be used. That is, every Small_Int variable is, in a sense, an Integer variable. The definition of subtype clearly disallows having public entities in the parent class that are not public in the subclass.

A subtype inherits interfaces and behavior, while a subclass inherits implementation, primarily to promote code reuse.

Single and Multiple Inheritance:

The purpose of multiple inheritance is to allow a new class to inherit from two or more classes.

Complexity of multiple inheritance is explained by several problems: First, note that if a class has two unrelated parent classes and neither defines a name that is defined in the other, there is no problem. However, suppose a subclass named C inherits from both class A and class B and both A and B define an inheritable method named display. If C needs to reference both versions of display which produces confusion.

Another issue arises if both A and B are derived from a common parent, Z, and C has both A and B as parent classes. This situation is called diamond or shared inheritance. In this case, both A and B should include Z's inheritable variables. Suppose Z includes an inheritable variable named sum. The question is whether C should inherit both versions of sum or just one, and if just one, which one? There may be programming situations in which just one of the two should be inherited, and others in which both should be inherited. A similar problem occurs when both A and B inherit a method from Z and both override that method. If a client of C, which inherits both overriding methods, calls the method, which method is called, or are both supposed to be called. Diamond inheritance is shown in Figure 12.3.

Downloaded From EnggTree.com

EnggTree.com

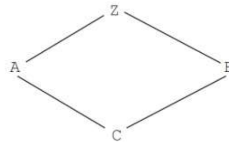


Figure 12.3
An example of diamond inheritance

The use of multiple inheritance can easily lead to complex program organizations. Many who have attempted to use multiple inheritance have found that designing the classes to be used as multiple parents is difficult.

An interface is somewhat similar to an abstract class; its methods are declared but not defined. Interfaces cannot be instantiated. They are used as an alternative to multiple inheritance.8 Interfaces provide some of the benefits of multiple inheritance but have fewer disadvantages. For example, the problems of diamond inheritance are avoided when interfaces, rather than multiple inheritance, are used.

Allocation and Deallocation of Objects:

There are two design questions concerning the allocation and deallocation of objects. The first of these is the place from which objects are allocated. If they behave like the abstract data types, then they can be allocated from anywhere. This means they could be allocated from the run-time stack or explicitly created on the heap with an operator or function, such as new. If they are all heap dynamic, there is the advantage of having a uniform method of creation and access through pointer or reference variables.

If objects are stack dynamic, there is a potential problem with regard to subtypes. If class B is a child of class A and B is a subtype of A, then an object of B type can be assigned to a variable of A type. For example, if b1 is a variable of B type and a1 is a variable of A type, then

a1 = b1;

is a legal statement. If a1 and b1 are references to heap-dynamic objects, there is no problem— the assignment is a simple pointer assignment. However, if a1 and b1 are stack dynamic, then they are value variables and, if assigned the value of the object, must be copied to the space of the target object. If B adds a data field to what it inherited from A, then a1 will not have sufficient space on the stack for all of b1. The excess will simply be truncated, which could be confusing to programmers who write or use the code. This truncation is called object slicing.

Downloaded From EnggTree.com

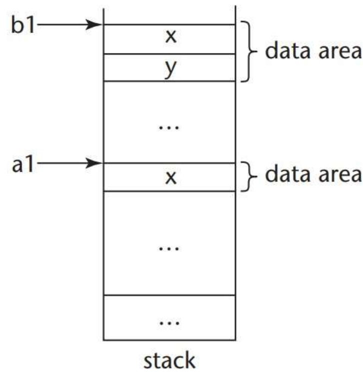


Figure 12.4

An example of object slicing

```
class A {
 int x;
 . . .
};
class B : A {
 int y;
 . . .
}
```

The second question is whether deallocation is implicit, explicit, or both.

Dynamic or Static Binding:

The question here is whether all bindings of messages to methods are dynamic. The alternative is to allow the user to specify whether a specific binding is to be dynamic or static.
Nested Classes:

One of the primary motivations for nesting class definitions is information hiding. If a new class is needed by only one class, there is no reason to define it so it can be seen by other classes. In this situation, the new class can be nested inside the class that uses it.
The class in which the new class is nested is called the nesting class.

Initialization of Object:

One question is whether objects must be initialized manually or through some implicit mechanism.

IMPLEMENTATION OF OBJECT ORIENTED CONSTRUCTS:

There are two parts of language support for object-oriented programming for language implementers: storage structures for instance variables and the dynamic bindings of messages to methods.

Instance Data Storage:

In C++, classes are defined as extensions of C’s record structures— structs. This similarity suggests a storage structure for the instance variables of class instances—that of a record. This form of this structure is called a class instance record (CIR). The structure of a CIR is static, so it is built at compile time.

Downloaded From EnggTree.com

Because the structure of the CIR is static, access to all instance variables can be done as it is in records, using constant offsets from the beginning of the CIR instance.
Dynamic Binding of Method Calls to Methods:

Methods in a class that are statically bound need not be involved in the CIR for the class. However, methods that will be dynamically bound must have entries in this structure. Such entries could simply have a pointer to the code of the method, which must be set at object creation time. Calls to a method could then be connected to the corresponding code through this pointer in the CIR. The drawback to this technique is that every instance would need to store pointers to all dynamically bound methods that could be called from the instance.

The list of dynamically bound methods that can be called from an instance of a class is the same for all instances of that class. So the CIR for an instance needs only a single pointer to that list to enable it to find called methods. The storage structure for the list is often called a virtual method table (vtable).

```
public class A {
 public int a, b;
 public void draw() { . . . }
 public int area() { . . . }
}
public class B extends A {
 public int c, d;
 public void draw() { . . . }
 public void sift() { . . . }
}
```

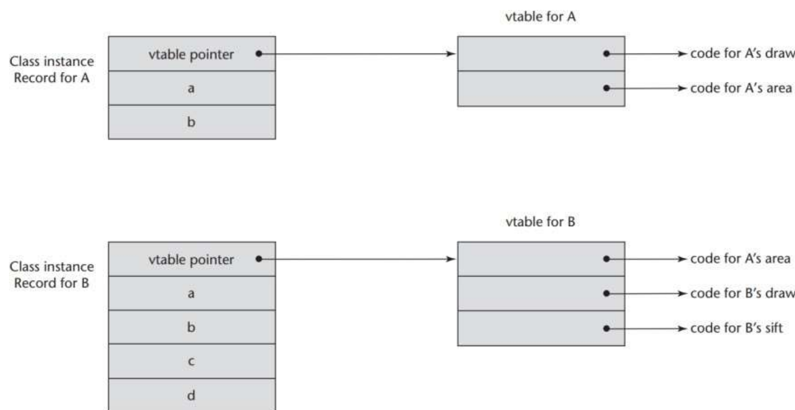


Figure 12.7

An example of the CIRs with single inheritance

Multiple inheritance complicates the implementation of dynamic binding. Consider the following three C++ class definitions

```
class A {
 public:
 int a;
 virtual void fun() { . . . }
 virtual void init() { . . . }
};
class B {
```

Downloaded From EnggTree.com

EnggTree.com

```
 public:
 int b;
 virtual void sum() { . . . }
};
class C : public A, public B {
 public:
 int c;
 virtual void fun() { . . . }
 virtual void dud() { . . . }
};
```

The C class inherits the variable a and the init method from the A class. It redefines the fun method, although both its fun and that of the parent class A are potentially visible through a polymorphic variable (of type A). From B, C inherits the variable b and the sum method. C defines its own variable, c, and defines an uninherited method, dud. A CIR for C must include A's data, B's data, and C's data, as well as some means of accessing all visible methods. Under single inheritance, the CIR would include a pointer to a vtable that has the addresses of the code of all visible methods. With multiple inheritance, however, it is not that simple. There must be at least two different views available in the CIR—one for each of the parent classes, one of which includes the view for the subclass, C. This inclusion of the view of the subclass in the parent class's view is just as in the implementation of single inheritance.

There must also be two vtables: one for the A and C view and one for the B view. The first part of the CIR for C in this case can be the C and A view, which begins with a vtable pointer for the methods of C and those inherited from A, and includes the data inherited from A. Following this in C's CIR is the B view part, which begins with a vtable pointer for the virtual methods of B, which is followed by the data inherited from B and the data defined in C. The CIR for C is shown in Figure 12.8

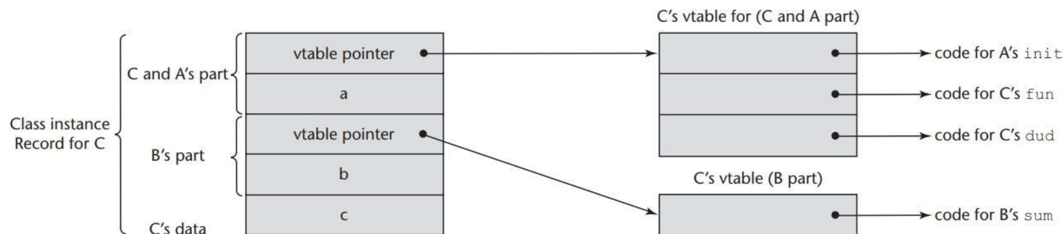


Figure 12.8

An example of a subclass CIR with multiple parents

CONCURRENCY:

Concurrency in software execution can occur at four different levels: instruction level (executing two or more machine instructions simultaneously), statement level (executing two or more high-level language statements simultaneously), unit level (executing two or more subprogram units simultaneously), and program level (executing two or more programs simultaneously).

Downloaded From EnggTree.com

Categories of Concurrency:

There are two distinct categories of concurrent unit control. The most natural category of concurrency is that in which, assuming that more than one processor is available, several program units from the same program literally execute simultaneously. This is physical concurrency.

A slight relaxation of this concept of concurrency allows the programmer and the application software to assume that there are multiple processors providing actual concurrency, when in fact the actual execution of programs is taking place in interleaved fashion on a single processor. This is logical concurrency.

Synchronization is a mechanism that controls the order in which tasks execute. Two kinds of synchronization are required when tasks share data: cooperation and competition. Cooperation synchronization is required between task A and task B when task A must wait for task B to complete some specific activity before task A can begin or continue its execution. Competition synchronization is required between two tasks when both require the use of some resource that cannot be simultaneously used.

The most important design issues for language support for concurrency have already been discussed at length: competition and cooperation synchronization. The following sections discuss three alternative approaches to the design issues for concurrency: semaphores, monitors, and message passing.

SEMAPHORES:

A semaphore is a simple mechanism that can be used to provide synchronization of tasks. Semaphores can also be used to provide cooperation synchronization. To provide limited access to a data structure, guards can be placed around the code that accesses the structure. A guard is a linguistic device that allows the guarded code to be executed only when a specified condition is true. a guard can be used to allow only one task to access a particular shared data structure at a time. A semaphore is an implementation of a guard. Specifically, a semaphore is a data structure that consists of an integer and a queue that stores task descriptors. A task descriptor is a data structure that stores all of the relevant information about the execution state of a task.

The only two operations provided for semaphores were originally named P and V by Dijkstra, after the two Dutch words passeren (to pass) and vrygeren (to release).

Cooperation Synchronization:

For cooperation synchronization, such a buffer must have some way of recording both the number of empty positions and the number of filled positions in the buffer (to prevent buffer underflow and overflow). The counter component of a semaphore can be used for this purpose. One semaphore variable—for example, emptyspots—can use its counter to maintain the number of empty locations in a shared buffer used by producers and consumers, and another, fullspots—can use its counter to maintain the number of filled locations in the buffer.

The queues of these semaphores can store the descriptors of tasks that have been forced to wait for access to the buffer. The queue of emptyspots can store producer tasks that are waiting for available positions in the buffer; the queue of fullspots can store consumer tasks waiting for values to be placed in the buffer.

Our example buffer is designed as an abstract data type in which all data enters the buffer through the subprogram DEPOSIT, and all data leaves the buffer through the

Downloaded From EnggTree.com

EnggTree.com

subprogram FETCH. The DEPOSIT subprogram needs only to check with the emptyspots semaphore to see whether there are any empty positions. If there is at least one, it can proceed with the DEPOSIT, which must have the side effect of decrementing the counter of emptyspots. If the buffer is full, the caller to DEPOSIT must be made to wait in the emptyspots queue for an empty spot to become available. When the DEPOSIT is complete, the DEPOSIT subprogram increments the counter of the fullspots semaphore to indicate that there is one more filled location in the buffer. The FETCH subprogram has the opposite sequence of DEPOSIT. It checks the fullspots semaphore to see whether the buffer contains at least one item. If it does, an item is removed and the emptyspots semaphore has its counter incremented by 1. If the buffer is empty, the calling task is put in the fullspots queue to wait until an item appears. When FETCH is finished, it must increment the counter of emptyspots.

The operations on semaphore types often are not direct—they are done through wait and release subprograms. Therefore, the DEPOSIT operation just described is actually accomplished in part by calls to wait and release. Note that wait and release must be able to access the task-ready queue. The wait semaphore subprogram is used to test the counter of a given semaphore variable. If the value is greater than zero, the caller can carry out its operation. In this case, the counter value of the semaphore variable is decremented to indicate that there is now one fewer of whatever it counts. If the value of the counter is zero, the caller must be placed on the waiting queue of the semaphore variable, and the processor must be given to some other ready task. The release semaphore subprogram is used by a task to allow some other task to have one of whatever the counter of the specified semaphore variable counts. If the queue of the specified semaphore variable is empty, which means no task is waiting, release increments its counter (to indicate there is one more of whatever is being controlled that is now available). If one or more tasks are waiting, release moves one of them from the semaphore queue to the ready queue.

The following are concise pseudocode descriptions of wait and release:

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
decrement aSemaphore's counter
else
 put the caller in aSemaphore's queue
 attempt to transfer control to some ready task
 (if the task-ready queue is empty, deadlock occurs)
end if
release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting)then
 increment aSemaphore's counter
else
 put the calling task in the task-ready queue
 transfer control to a task from aSemaphore's queue
end
```

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
```

Downloaded From EnggTree.com

EnggTree.com

```
task producer;
 loop
 -- produce VALUE --
 wait(emptyspots); { wait for a space }
 DEPOSIT(VALUE);
 release(fullspots); { increase filled spaces }
 end loop;
end producer;
task consumer;
loop
 wait(fullspots); { make sure it is not empty }
FETCH(VALUE);
 release(emptyspots); { increase empty spaces }
 -- consume VALUE --
 end loop
end consumer;
```

Competition Synchronization:

Access to the structure can be controlled with an additional semaphore. This semaphore need not count anything but can simply indicate with its counter whether the buffer is currently being used. The wait statement allows the access only if the semaphore's counter has the value 1, which indicates that the shared buffer is not currently being accessed. If the semaphore's counter has a value of 0, there is a current access taking place, and the task is placed in the queue of the semaphore. Notice that the semaphore's counter must be initialized to 1.

A semaphore that requires only a binary-valued counter, like the one used to provide competition synchronization in the following example, is called a binary semaphore.

The access semaphore is used to ensure mutually exclusive access to the buffer.There may be more than one producer and more than one consumer.

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
 loop
 -- produce VALUE --
 wait(emptyspots); { wait for a space }
 wait(access); { wait for access }
 DEPOSIT(VALUE);
 release(access); { relinquish access }
 release(fullspots); { increase filled spaces }
 end loop;
end producer;

task consumer;
 loop
```

Downloaded From EnggTree.com

EnggTree.com

```
 wait(fullspots); { make sure it is not empty }
 wait(access); { wait for access }
 FETCH(VALUE);
 release(access); { relinquish access }
 release(emptyspots); { increase empty spaces }
 -- consume VALUE --
 end loop
end consumer;
```

Evaluation:

Using semaphores to provide cooperation synchronization creates an unsafe programming environment. In the buffer example, leaving out the wait(emptyspots) statement of the producer task would result in buffer overflow. Leaving out the wait(- fullspots) statement of the consumer task would result in buffer underflow. Leaving out either of the releases would result in deadlock. These are cooperation synchronization failures.

The reliability problems that semaphores cause in providing cooperation synchronization also arise when using them for competition synchronization. Leaving out the wait(access) statement in either task can cause insecure access to the buffer. Leaving out the release(access) statement in either task results in deadlock. These are competition synchronization failures.

MONITORS:

One solution to some of the problems of semaphores in a concurrent environment is to encapsulate shared data structures with their operations and hide their representations—that is, to make shared data structures abstract data types with some special restrictions. This solution can provide competition synchronization without semaphores by transferring responsibility for synchronization to the run-time system.

Competition Synchronization:

One of the most important features of monitors is that shared data is resident in the monitor rather than in any of the client units. The programmer does not synchronize mutually exclusive access to shared data through the use of semaphores or other mechanisms. Calls to monitor procedures are implicitly blocked and stored in a queue if the monitor is busy at the time of the call.

Cooperation Synchronization:

Although mutually exclusive access to shared data is intrinsic with a monitor, cooperation between processes is still the task of the programmer. In particular, the programmer must guarantee that a shared buffer does not experience underflow or overflow. A program containing four tasks and a monitor that provides synchronized access to a concurrently shared buffer is shown in Figure.

Downloaded From EnggTree.com

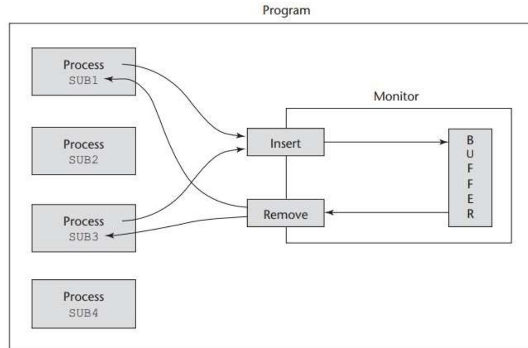


Figure 13.3
A program using a monitor to control access to a shared buffer

The interface to the monitor is shown as the two boxes labeled insert and remove. The monitor appears exactly like an abstract data type—a data structure with limited access—which is what a monitor is.

Evaluation:

Monitors are a better way to provide competition synchronization than are semaphores, primarily because of the problems of semaphores.

Semaphores and monitors are equally powerful at expressing concurrency control—semaphores can be used to implement monitors and monitors can be used to implement semaphores.

MESSAGE PASSING:

The Concept of Synchronous Message Passing:

Message passing can be either synchronous or asynchronous. The basic concept of synchronous message passing is that tasks are often busy, and when busy, they cannot be interrupted by other units. Suppose task A and task B are both in execution, and A wishes to send a message to B. Clearly, if B is busy, it is not desirable to allow another task to interrupt it. That would disrupt B's current processing. The alternative is to provide a linguistic mechanism that allows a task to specify to other tasks when it is ready to receive messages.

A task can be designed so that it can suspend its execution at some point, either because it is idle or because it needs information from another unit before it can continue. If task A is waiting for a message at the time task B sends that message, the message can be transmitted. This actual transmission of the message is called a rendezvous. Note that a rendezvous can occur only if both the sender and receiver want it to happen. During a rendezvous, the information of the message can be transmitted in either or both directions.

THREADS:

JAVA THREADS:

The Thread class is not the natural parent of any other classes. It provides some services for its subclasses, but it is not related in any natural way to their computational purposes. Thread is the only class available for creating concurrent Java programs.

The Thread class includes five constructors and a collection of methods and constants. The run method, which describes the actions of the thread, is always overridden by subclasses of Thread. The start method of Threadstarts its thread as a concurrent unit by calling its run method.

The call to start is unusual in that control returns immediately to the caller, which then continues its execution, in parallel with the newly started run method.

Downloaded From EnggTree.com

Following is a skeletal subclass of Thread and a code fragment that creates an object of the subclass and starts the run method's execution in the new thread:

```
class MyThread extends Thread {
 public void run() { . . . }
}
. . .
Thread myTh = new MyThread();
myTh.start()    ;
```

When a Java application program begins execution, a new thread is created (in which the main method will run) and main is called. Therefore, all Java application programs run in threads. When a program has multiple threads,a scheduler must determine which thread or threads will run at any given time. The Thread class provides several methods for controlling the execution of threads. The yield method, which takes no parameters, is a request from the running thread to surrender the processor voluntarily.

The sleep method has a single parameter, which is the integer number of milliseconds that the caller of sleep wants the thread to be blocked. After the specified number of milliseconds has passed, the thread will be put in the task-ready queue. Because there is no way to know how long a thread will be in the task-ready queue before it runs, the parameter to sleep is the minimum amount of time the thread will not be in execution. The sleep method can throw an InterruptedException, which must be handled in the method that calls sleep.

The join method is used to force a method to delay its execution until the run method of another thread has completed its execution. join is used when the processing of a method cannot continue until the work of the other thread is complete.

```
public void run() {
 . . .
 Thread myTh = new Thread();
 myTh.start();
 // do part of the computation of this thread
 myTh.join(); // Wait for myTh to complete
 // do the rest of the computation of this thread
}
```

The priorities of threads need not all be the same. A thread's default priority initially is the same as the thread that created it. If main creates a thread, its default priority is the constant NORM_PRIORITY, which is usually 5. Thread defines two other priority constants, MAX_PRIORITY and MIN_PRIORITY, whose values are usually 10 and 1, respectively. The priority of a thread can be changed with the method setPriority. The new priority can be any of the predefined constants or any other number between MIN_PRIORITY and MAX_PRIORITY. The getPriority method returns the current priority of a thread. The priority constants are defined in Thread.

Downloaded From EnggTree.com

SEMAPHORE:

The java.util.concurrent.Semaphore package defines the Semaphore class. Objects of this class implement counting semaphores. A counting semaphore has a counter, but no queue for storing thread descriptors. The Semaphore class defines two methods, acquire and release, which correspond to the wait and release operations. The basic constructor for Semaphore takes one integer parameter, which initializes the semaphore's counter. For example, the following could be used to initialize the fullspots and emptyspots semaphores for the buffer example

fullspots = new Semaphore(0);

emptyspots = new Semaphore(BUFLEN);

The deposit operation of the producer method would appear as follows:

emptyspots.acquire();

deposit(value);

fullspots.release();

Likewise, the fetch operation of the consumer method would appear as follows:

fullspots.acquire();

fetch(value);

emptyspots.release();

Competition Synchronization:

Competition synchronization on an object is implemented by specifying that the methods that access shared data are synchronized. The synchronized mechanism is implemented as follows: Every Java object has a lock. Synchronized methods must acquire the lock of the object before they are allowed to execute, which prevents other synchronized methods from executing on the object during that time. A synchronized method releases the lock on the object on which it runs when it completes its execution, even if that completion is due to an exception. Consider the following skeletal class definition:

```
class ManageBuf {
 private int [100] buf;
 . . .
 public synchronized void deposit(int item) { . . . }
 public synchronized int fetch() { . . . }
 . . .
}
```

Downloaded From EnggTree.com

Cooperation Synchronization:

Cooperation synchronization in Java is implemented with the wait, notify, and notifyAll methods, all of which are defined in Object, the root class of all Java classes. All classes except Object inherit these methods. Every object has a wait list of all of the threads that have called wait on the object. The notify method is called to tell one waiting thread that an event that it may have been waiting for has occurred. The specific thread that is awakened by notify cannot be determined, because the Java Virtual Machine (JVM) chooses one from the wait list of the thread object at random. Because of this, along with the fact that the waiting threads may be waiting for different conditions, the notifyAll method is often used, rather than notify. The notifyAll method awakens all of the threads on the object's wait list by putting them in the task-ready queue.

STATEMENT LEVEL CONCURRENCY:

The objective of such designs is to provide a mechanism that the programmer can use to inform the compiler of ways it can map the program onto a multiprocessor architecture.

High-Performance Fortran:

High-Performance Fortran (HPF; ACM, 1993b) is a collection of extensions to Fortran 90 that are meant to allow programmers to specify information to the compiler to help it optimize the execution of programs on multiprocessor computers. HPF includes both new specification statements and intrinsic, or built-in, subprograms.

The primary specification statements of HPF are for specifying the number of processors, the distribution of data over the memories of those processors, and the alignment of data with other data in terms of memory placement. The HPF specification statements appear as special comments in a Fortran program. Each of them is introduced by the prefix !HPF$, where ! is the character used to begin lines of comments in Fortran 90. This prefix makes them invisible to Fortran 90 compilers but easy for HPF compilers to recognize.

The PROCESSORS specification has the following form:

!HPF$ PROCESSORS procs (n)

This statement is used to specify to the compiler the number of processors that can be used by the code generated for this program.

The DISTRIBUTE and ALIGN specifications are used to provide information to the compiler on machines that do not share memory—that is, each processor has its own memory.

The DISTRIBUTE statement specifies what data are to be distributed and the kind of distribution that is to be used. Its form is as follows:

!HPF$ DISTRIBUTE (kind) ONTO procs :: identifier_list

In this statement, kind can be either BLOCK or CYCLIC. The identifier list is the names of the array variables that are to be distributed. A variable that is specified to be BLOCK distributed is divided into n equal groups, where each group consists of contiguous collections of array elements evenly distributed over the memories of all the processors. A CYCLIC distribution specifies that individual elements of the array are cyclically stored in the memories of the processors.

The form of the ALIGN statement is

ALIGN array1_element WITH array2_element

ALIGN is used to relate the distribution of one array with that of another. For example,

ALIGN list1(index) WITH list2(index+1)

Downloaded From EnggTree.com

specifies that the index element of list1 is to be stored in the memory of the same processor as the index+1 element of list2, for all values of index.
Consider the following example code segment:

```
 REAL list_1 (1000), list_2 (1000)
 INTEGER list_3 (500), list_4 (501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: list_1, list_2
!HPF$ ALIGN list_3 (index) WITH list_4 (index+1)
 . . .
 list_1 (index) = list_2 (index)
 list_3 (index) = list_4 (index+1)
```

In each execution of these assignment statements, the two referenced array elements will be stored in the memory of the same processor.
The FORALL statement specifies a sequence of assignment statements that may be executed concurrently. For example,

```
FORALL (index = 1:1000) list_1(index) = list_2(index) END FORALL
```

specifies the assignment of the elements of list_2 to the corresponding elements of list_1. However, the assignments are restricted to the following order: the right side of all 1,000 assignments must be evaluated first, before any assignments take place.

EXCEPTION HANDLING:
Exception is an unusual event, erroneous or not, that is detectable by either hardware or software and that may require special processing. The special processing that may be required when an exception is detected is called exception handling. This processing is done by a code unit or segment called an exception handler. An exception is raised when its associated event occurs. Different kinds of exceptions require different exception handlers.
The absence of separate or specific exception-handling facilities in a language does not preclude the handling of user-defined, software-detected exceptions. Such an exception detected within a program unit is often handled by the unit's caller.

If it is desirable to handle an exception in the unit in which it is detected, the handler is included as a segment of code in that unit. There are some definite advantages to having exception handling built into a language. First, without exception handling, the code required to detect error conditions can considerably clutter a program.
For example,
consider the following reference to an element of mat, which has 10 rows and 20 columns:

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
 sum += mat[row][col];
else
 System.out.println("Index range error on mat, row = " + row + " col = " + col);
```

The presence of exception handling in the language would permit the compiler to insert machine code for such checks before every array element access, greatly shortening and simplifying the source program. Another advantage of language support for exception handling results from exception propagation. Exception propagation allows an exception

Downloaded From EnggTree.com

raised in one program unit to be handled in some other unit in its dynamic or static ancestry. This allows a single exception handler to be used for any number of different program units.

A language that supports exception handling encourages its users to consider all of the events that could occur during program execution and how they can be handled.

DESIGN ISSUES:

Some of the design issues for an exception-handling system when it is part of a programming language. Such a system might allow both predefined and user- defined exceptions and exception handlers. Note that predefined exceptions are implicitly raised, whereas user-defined exceptions must be explicitly raised by user code. Consider the following skeletal subprogram that includes an exception-handling mechanism for an implicitly raised exception:

```
 void example() {
. . .
average = sum / total;
. . . return;
 /* Exception handlers */
when zero_divide {
average = 0;
printf("Error–divisor (total) is zero\n"); } . . . }
```

The exception of division by zero, which is implicitly raised, causes control to transfer to the appropriate handler, which is then executed. The first design issue for exception handling is how an exception occurrence is bound to an exception handler. This issue occurs on two different levels. On the unit level, there is the question of how the same exception being raised at different points in a unit can be bound to different handlers within the unit. For example, in the example subprogram, there is a handler for a division- by- zero exception that appears to be written to deal with an occurrence of division by zero in a particular statement (the one shown). But suppose the function includes several other expressions with division operators. For those operators, this handler would probably not be appropriate. So, it should be possible to bind the exceptions that can be raised by particular statements to particular handlers, even though the same exception can be raised by many different statements.

After an exception handler executes, either control can transfer to somewhere in the program outside of the handler code or program execution can simply terminate. We term this the question of control continuation after handler execution, or simply continuation. Termination is obviously the simplest choice, and in many error exception conditions, the best. However, in other situations, particularly those associated with unusual but not erroneous events, the choice of continuing execution is best. This design is called resumption.

When exception handling is included, a subprogram's execution can terminate in two ways: when its execution is complete or when it encounters an exception.

Downloaded From EnggTree.com

EnggTree.com

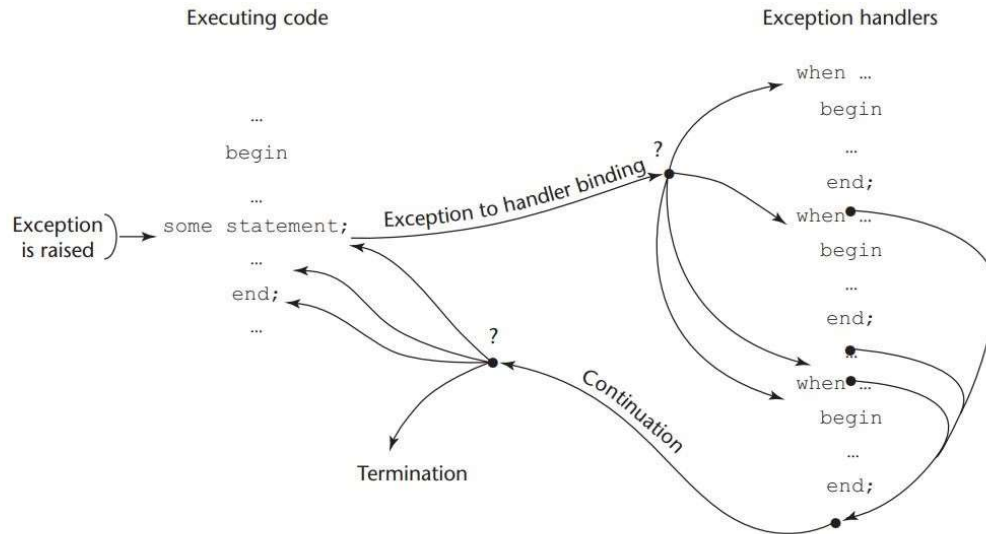


Figure 14.1

Exception-handling control flow

In some situations, it is necessary to complete some computation, regardless of how subprogram execution terminates. The ability to specify such a computation is called finalization.

The exception-handling design issues can be summarized as follows:

- How and where are exception handlers specified, and what is their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about an exception be passed to the handler?

Where does execution continue, if at all, after an exception handler completes its execution? (This is the question of continuation or resumption.)

- Is some form of finalization provided?
- How are user-defined exceptions specified?
- If there are predefined exceptions, should there be default exception handlers for programs that do not provide their own?
- Can predefined exceptions be explicitly raised?
- Are hardware- detectable errors treated as exceptions that may be handled?
- Are there any predefined exceptions?

EXCEPTION HANDLING IN C++:

C++ uses a special construct that is introduced with the reserved word try to specify the scope for exception handlers. A try construct includes a compound statement called the try clause and a list of exception handlers. The compound statement defines the scope of the following handlers. The general form of this construct is as follows:

```
try {
//** Code that might raise an exception
} catch(formal parameter) {
//** A handler body
}
```

Downloaded From EnggTree.com

```
. . .
catch(formal parameter) {
//** A handler body
}
```

Each catch function is an exception handler. A catch function can have only a single formal parameter, which is similar to a formal parameter in a function definition in C++.

BINDING EXCEPTIONS:

C++ exceptions are raised only by the explicit statement throw, whose general form is as follows:

```
 throw [expression];
```

The brackets here are metasymbols used to specify that the expression is optional. A throw without an operand can appear only in a handler. When it appears there, it reraises the exception, which is then handled elsewhere. The type of the throw expression selects the particular handler, which of course must have a "matching" type formal parameter. In this case, matching means the following: A handler with a formal parameter of type T, const T, T& (a reference to an object of type T), or const T& matches a throw with an expression of type T. In the case where T is a class, a handler whose parameter is type T or any class that is an ancestor of T matches.

EXCEPTION HANDLING IN JAVA:

Java's exception handling is based on that of C++, but it is designed to be more in line with the object-oriented language paradigm.

Classes of Exceptions:

All Java exceptions are objects of classes that are descendants of the Throwable class. The Java system includes two predefined exception classes that are subclasses of Throwable: Error and Exception. The Error class and its descendants are related to errors that are thrown by the Java run-time system, such as running out of heap memory. These exceptions are never thrown by user programs, and they should never be handled there. There are two system-defined direct descendants of Exception: RuntimeException and IOException.

Exception Handlers:

The exception handlers of Java have the same form as those of C++, except that every catch must have a parameter and the class of the parameter must be a descendant of the predefined class Throwable.

Binding Exceptions to Handlers:

Throwing an exception is simple. An instance of the exception class is given as the operand of the throw statement. For example, suppose we define an exception named MyException as class MyException extends Exception {

```
public MyException() {}
 public MyException(String message) {
 super (message);
 }
}
```

This exception can be thrown with the following statement: throw new MyException();

Downloaded From EnggTree.com

EnggTree.com

The binding of exceptions to handlers in Java is similar to that of C++. If an exception is thrown in the compound statement of a try construct, it is bound to the first handler (catch function) immediately following the try clause whose parameter is the same class as the thrown object, or an ancestor of it. If a matching handler is found, the throw is bound to it and it is executed.

The finally Clause:

There are some situations in which a process must be executed regardless of whether a try clause throws an exception or the exception is handled in the method. A finally clause is placed at the end of the list of handlers just after a complete try construct. try {

```
 . . .
}
catch (. . .) {
 . . .
}
. . . //** More handlers
finally {
 . . .
}
```

The semantics of this construct is as follows: If the try clause throws no exceptions, the finally clause is executed before execution continues after the try construct.

Assertions:

There are two possible forms of the assert statement:

assert condition;

assert condition : expression;

In the first case, the condition is tested when execution reaches the assert. If the condition evaluates to true, nothing happens. If it evaluates to false, the AssertionError exception is thrown. In the second case, the action is the same, except that the value of the expression is passed to the AssertionError constructor as a string and beomes debugging output.

EVENT HANDLING:

An event is a notification that something specific has occurred, such as a mouse click on a graphical button. Strictly speaking, an event is an object that is implicitly created by the run-time system in response to a user action, at least in the context in which event handling is being discussed here.

An event handler is a segment of code that is executed in response to the appearance of an event. Event handlers enable a program to be responsive to user actions.

JAVA:

In late 1998, a new collection of components was added. These were collectively called Swing.

The Swing collection of classes and interfaces, defined in javax.swing, includes GUI components, or widgets.

A text box is an object of class JTextField. The simplest JTextField constructor takes a single parameter, the length of the box in characters. For example,

JTextField name = new JTextField(32);

Downloaded From EnggTree.com

EnggTree.com

Radio buttons are special buttons that are placed in a button group container. A button group is an object of class ButtonGroup, whose constructor takes no parameters. In a radio button group, only one button can be pressed at a time. If any button in the group becomes pressed, the previously pressed button is implicitly unpressed.

```
ButtonGroup payment = new ButtonGroup();
JRadioButton box1 = new JRadioButton("Visa", true);
JRadioButton box2 = new JRadioButton("Master Charge");
JRadioButton box3 = new JRadioButton("Discover");
payment.add(box1);
payment.add(box2);
payment.add(box3);
```

Java Event Model:

When a user interacts with a GUI component, for example by clicking a button, the component creates an event object and calls an event handler through an object called an event listener, passing the event object. The event handler provides the associated actions. GUI components are event generators. In Java, events are connected to event handlers through event listeners. Event listeners are connected to event generators through event listener registration. Listener registration is done with a method of the class that implements the listener interface, as described later in this section. Only event listeners that are registered for a specific event are notified when that event occurs. The listener method that receives the message implements an event handler. To make the event-handling methods conform to a standard protocol, an interface is used. An interface prescribes standard method protocols but does not provide implementations of those methods.

All the event-related classes are in the java.awt.event package, so it is imported to any class that uses events

C#:

Event handling in C# is similar to that of Java. NET provides two approaches to creating GUIs in applications, the original Windows Forms and the more recent Windows Presentation Foundation.

Using Windows Forms, a C# application that constructs a GUI is created by subclassing the Form predefined class, which is defined in the System.Windows.Forms namespace. This class implicitly provides a window to contain our components.

Text can be placed in a Label object and radio buttons are objects of the RadioButton class. The size of a Label object is not explicitly specified in the constructor; rather it can be specified by setting the AutoSize data member of the Label object to true, which sets the size according to what is placed in it. Components can be placed at a particular location in the window by assigning a new Point object to the Location property of the component. The Point class is defined in the System.Drawing namespace. The Point constructor takes two parameters, which are the coordinates of the object in pixels. For example, Point(100, 200) is a position that is 100 pixels from the left edge of the window and 200 pixels from the top.

```
private RadioButton plain = new RadioButton();
plain.Location = new Point(100, 300);
plain.Text = "Plain";
Controls.Add(plain)
```

Downloaded From EnggTree.com

All C# event handlers have the same protocol: the return type is void and the two parameters are of types object and EventArgs. Neither of the parameters needs to be used for a simple situation. An event handler method can have any name. A radio button is tested to determine whether it is clicked with the Boolean Checked property of the button. Consider the following skeletal example of an event handler:

```
private void rb_CheckedChanged (object o, EventArgs e){
 if (plain.Checked) . . .
 . . .
}
```

Downloaded From EnggTree.com

UNIT 5

FUNCTIONAL AND LOGIC PROGRAMMING LANGUAGES

INTRODUCTION TO LAMBDA CALCULUS:

A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set. A function definition specifies the domain and range sets, either explicitly or implicitly, along with the mapping. The mapping is described by an expression or, in some cases, by a table.

One of the fundamental characteristics of mathematical functions is that the evaluation order of their mapping expressions is controlled by recursion and conditional expressions, rather than by the sequencing and iterative repetition that are common to programs written in the imperative programming languages.

Another important characteristic of mathematical functions is that because they have no side effects and cannot depend on any external values, they always map a particular element of the domain to the same element of the range.

Function definitions are often written as a function name, followed by a list of parameters in parentheses, followed by the mapping expression. For example,

cube(x) ≡ x * x * x,

where x is a real number. In this definition, the domain and range sets are the real numbers. The symbol ≡ is used to mean "is defined as." The parameter x can represent any member of the domain set, but it is fixed to represent one specific element during evaluation of the function expression.

cube (2.0) = 2.0 * 2.0 * 2.0 = 8

The parameter x is bound to 2.0 during the evaluation and there are no unbound parameters. Furthermore, x is a constant (its value cannot be changed) during the evaluation.

A lambda expression specifies the parameters and the mapping of a function. The lambda expression is the function itself, which is nameless. For example, consider the following lambda expression:

λ(x)x * x * x

Church defined a formal computation model (a formal system for function definition, function application, and recursion) using lambda expressions. This is called lambda calculus. Lambda

Downloaded From EnggTree.com

calculus can be either typed or untyped. Untyped lambda calculus serves as the inspiration for the functional programming language..

Application of the example lambda expression is denoted as in the following example:

(λ(x)x * x * x)(2) which results in the value 8.

Functional form

A higher- order function, or functional form, is one that either takes one or more functions as parameters or yields a function as its result, or both. One common kind of functional form is function composition, which has two functional parameters and yields a function whose value is the first actual parameter function applied to the result of the second. Function composition is written as an expression, using ° as an operator, as in

h ≡ f ° g
For example, if
f(x) ≡ x + 2
g(x) ≡ 3*x
then h is defined as
h(x) ≡ f (g(x)), or h(x) ≡ (3*x) x+ 2

Apply-to-all is a functional form that takes a single function as a parameter. If applied to a list of parameters, apply-to-all applies its functional parameter to each of the values in the list parameter and collects the results in a list or sequence. Apply-to-all is denoted by a. Consider the following example:
Let
h(x) ≡ x*x
then
a(h,(2,3,4)) yields (4, 9, 16)

FUNDAMENTALS OF FUNCTIONAL PROGRAMMING LANGUAGES:
The objective of the design of a functional programming language is to mimic mathematical functions to the greatest extent possible. In an imperative language, an expression is evaluated and the result is stored in a memory location, which is represented as a variable in a program. This is the purpose of assignment statements. This necessary attention to memory cells, whose values represent the state of the program, results in a relatively low-level programming methodology. A program in an assembly language often must also store the results of partial evaluations of expressions. For example, to evaluate
(x + y)/(a - b)
the value of (x + y) is computed first. That value must then be stored while (a - b) is evaluated. The compiler handles the storage of intermediate results of expression evaluations in high-level

Downloaded From EnggTree.com

languages. The storage of intermediate results is still required, but the details are hidden from the programmer.

A purely functional programming language does not use variables or assignment statements, thus freeing the programmer from concerns related to the memory cells, or state, of the program. Without variables, iterative constructs are not possible, for they are controlled by variables. Repetition must be specified with recursion rather than with iteration. Programs are function definitions and function application specifications, and executions consist of evaluating function applications. Without variables, the execution of a purely functional program has no state in the sense of operational and denotational semantics. The execution of a function always produces the same result when given the same parameters. This feature is called referential transparency. It makes the semantics of purely functional languages far simpler than the semantics of the imperative languages.

A functional language provides a set of primitive functions, a set of functional forms to construct complex functions from those primitive functions, a function application operation, and some structure or structures for representing data. These structures are used to represent the parameters and values computed by functions. If a functional language is well designed, it requires only a relatively small number of primitive functions.

INTRODUCTION TO SCHEME:

The Scheme language is a language with simple syntax and semantics, Scheme is well suited to educational applications, such as courses in functional programming, and also to general introductions to programming.

The Scheme Interpreter:

A Scheme interpreter repeatedly reads an expression typed by the user (in the form of a list), interprets the expression, and displays the resulting value. This form of interpreter is also used by Ruby and Python. Expressions are interpreted by the function EVAL. Expressions that are calls to primitive functions are evaluated in the following way: First, each of the parameter expressions is evaluated, in no particular order. Then, the primitive function is applied to the parameter values, and the resulting value is displayed.

Comments in Scheme are any text following a semicolon on any line.

Primitive Numeric Functions:

Scheme includes primitive functions for the basic arithmetic operations. These are +, −, *, and /, for add, subtract, multiply, and divide. * and + can have zero or more parameters. If * is given no parameters, it returns 1; if + is given no parameters, it returns 0. + adds all of its parameters together. * multiplies all its parameters together. / and − can have two or more parameters.

Expression	Value
42	42
(* 3 7)	21
(+ 5 7 8)	20
(− 5 6)	−1

Downloaded From EnggTree.com

(− 15 7 2) 6
(− 24 (* 4 3)) 12

There are a large number of other numeric functions in Scheme, among them MODULO, ROUND, MAX, MIN, LOG, SIN, and SQRT. SQRT returns the square root of its numeric parameter, if the parameter's value is not negative. If the parameter is negative, SQRT yields a complex number.

Defining Functions:

In Scheme, a nameless function actually includes the word LAMBDA, and is called a lambda expression. For example,
(LAMBDA (x) (* x x))
is a nameless function that returns the square of its given numeric parameter. This function can be applied in the same way that named functions are: by placing it in the beginning of a list that contains the actual parameters. For example, the following expression yields 49:
((LAMBDA (x) (* x x)) 7)
In this expression, x is called a bound variable within the lambda expression. During the evaluation of this expression, x is bound to 7. A bound variable never changes in the expression after being bound to an actual parameter value at the time evaluation of the lambda expression begins.

The Scheme special form function DEFINE serves two fundamental needs of Scheme programming: to bind a name to a value and to bind a name to a lambda expression.
The simplest form of DEFINE is one used to bind a name to the value of an expression. This form is
(DEFINE symbol expression)
For example,
(DEFINE pi 3.14159)
(DEFINE two_pi (* 2 pi))
If these two expressions have been typed to the Scheme interpreter and then pi is typed, the number 3.14159 will be displayed; when two_pi is typed, 6.28318 will be displayed.
Names in Scheme can consist of letters, digits, and special characters except parentheses; they are case insensitive and must not begin with a digit.
The second use of the DEFINE function is to bind a lambda expression to a name. In this case, the lambda expression is abbreviated by removing the word LAMBDA. To bind a name to a lambda expression, DEFINE takes two lists as parameters. The first parameter is the prototype of a function call, with the function name followed by the formal parameters, together in a list. The second list contains an expression to which the name is to be bound. The general form of such a DEFINE is
(DEFINE (function_name parameters)

Downloaded From EnggTree.com

(expression)
)
The following example call to DEFINE binds the name square to a functional expression that takes one parameter:
(DEFINE (square number) (* number number))
After the interpreter evaluates this function, it can be used, as in
(square 5)
which displays 25.

Output Functions:

Scheme includes a few simple output functions, but when used with the interactive interpreter, most output from Scheme programs is the normal output from the interpreter, displaying the results of applying EVAL to top-level functions.

Numeric Predicate Function:

A predicate function is one that returns a Boolean value.

Function	Meaning
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?

Notice that the names for all predefined predicate functions that have words for names end with question marks. In Scheme, the two Boolean values are #T and #F (or #t and #f), although the Scheme predefined predicate functions return the empty list ,(),for false.

Control Flow:

Scheme uses three different constructs for control flow: one similar to the selection construct of the imperative languages and two based on the evaluation control used in mathematical functions. The Scheme two-way selector function, named IF, has three parameters: a predicate expression, a then expression, and an else expression. A call to IF has the form
(IF predicate then_expression else_expression)

```
(DEFINE (factorial n)
(IF (<= n 1)
1
 (* n (factorial (− n 1)))
```

Downloaded From EnggTree.com

EnggTree.com

))

List Functions:

One of the more common uses of the Lisp-based programming languages is list processing. This subsection introduces the Scheme functions for dealing with lists.

Suppose we have a function that has two parameters, an atom and a list, and the purpose of the function is to determine whether the given atom is in the given list. Neither the atom nor the list should be evaluated; they are literal data to be processed. To avoid evaluating a parameter, it is first given as a parameter to the primitive function QUOTE, which simply returns it without change. The following examples illustrate QUOTE:

(QUOTE A) returns A

(QUOTE (A B C)) returns (A B C)

Calls to QUOTE are usually abbreviated by preceding the expression to be quoted with an apostrophe (') and leaving out the parentheses around the expression. Thus, instead of (QUOTE (A B)), '(A B) is used.

The necessity of QUOTE arises because of the fundamental nature of Scheme data and code have the same

Following are additional examples of the operations of CAR and CDR:

(CAR '(A B C)) returns A

(CAR '((A B) C D)) returns (A B)

(CAR 'A) is an error because A is not a list

(CAR '(A)) returns A

(CAR '()) is an error

(CDR '(A B C)) returns (C)

(CDR '((A B) C D)) returns (C D)

(CDR 'A) is an error

(CDR '(A)) returns ()

(CDR '()) is an error.

Two machine instructions, also named CAR (contents of the address part of a register) and CDR (contents of the decrement part of a register), that extracted the associated fields.

As another example of a simple function, consider

(DEFINE (second a_list) (CAR (CDR a_list)))

Once this function is evaluated, it can be used, as in

(second '(A B C))

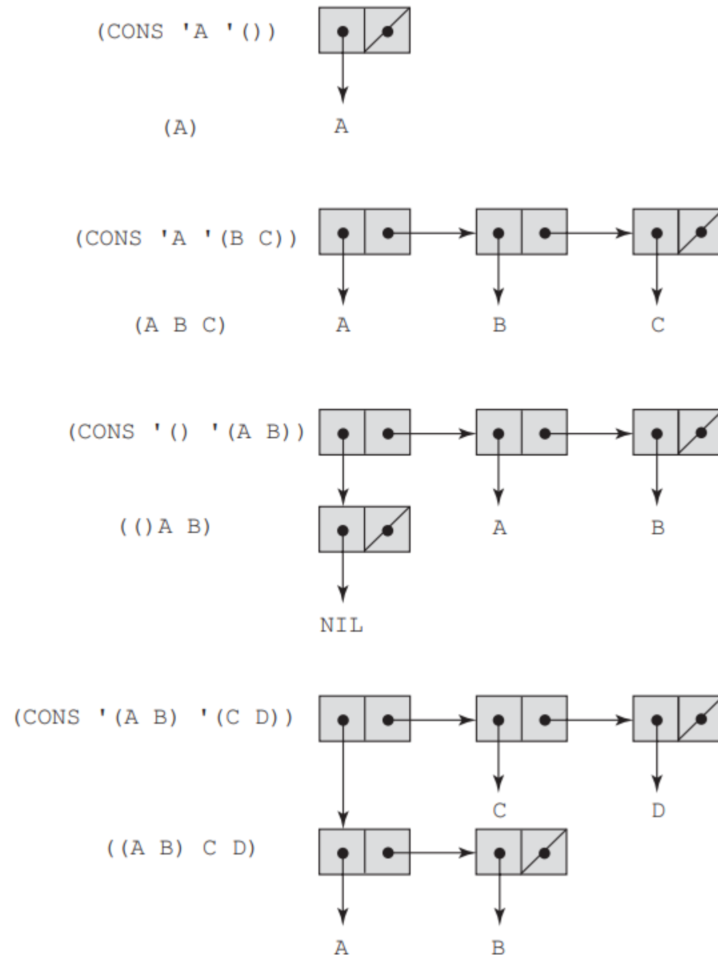
which returns B.

Downloaded From EnggTree.com

EnggTree.com

Figure 15.2

The result of several CONS operations



EXAMPLE SCHEME FUNCTIONS:

This section contains several examples of function definitions in Scheme. These programs solve simple list-processing problems.

Consider the problem of membership of a given atom in a given list that does not include sublists. Such a list is called a simple list. If the function is named member, it could be used as follows:

(member 'B '(A B C)) returns #T

(member 'B '(A C D E)) returns #F

Thinking in terms of iteration, the membership problem is simply to compare the given atom and the individual elements of the given list, one at a time in some order, until either a match is found or there are no more elements in the list to be compared. A similar process can be accomplished using recursion. The function can compare the given atom with the CAR of the list. If they match, the value #T is returned. If they do not match, the CAR of the list should be ignored and the search continued on the CDR of the list. This process will end if the given atom is found in the list. If the atom is not in the list, the function will eventually be called (by itself) with a null

Downloaded From EnggTree.com

list as the actual parameter. That event must force the function to return #F. Either the list is mpty on some call, in which case #F is returned, or a match is found and #T is returned. Altogether, there are three cases that must be handled in the function: an empty input list, a match between the atom and the CAR of the list, or a mismatch between the atom and the CAR of the list, which causes the recursive call.
These three are the three parameters to COND, with the last being the default case that is triggered by an ELSE predicate. The complete function follows:

```
(DEFINE (member atm a_list)
 (COND
 ((NULL? a_list) #F)
 ((EQ? atm (CAR a_list)) #T)
 (ELSE (member atm (CDR a_list)))
))
```

This form is typical of simple Scheme list-processing functions. In such functions, the data in lists are processed one element at a time. The individual elements are specified with CAR, and the process is continued using recursion on the CDR of the list.

LET

LET is a function that creates a local scope in which names are temporarily bound to the values of expressions. It is often used to factor out the common subexpressions from more complicated expressions.
The following example illustrates the use of LET. It computes the roots of a given quadratic equation, assuming the roots are real. The mathematical definitions of the real (as opposed to complex) roots of the quadratic equation ax2 + bx + c are as follows:
 root1 = (-b + sqrt(b2 - 4 ac))/2a and
root 2 = (-b - sqrt(b2 - 4 ac))/2 a

```
(DEFINE (quadratic_roots a b c)
 (LET (
 (root_part_over_2a
 (/ (SQRT (− (* b b) (* 4 a c))) (* 2 a)))
 (minus_b_over_2a (/ (− 0 b) (* 2 a)))
 )
 (LIST (+ minus_b_over_2a root_part_over_2a)
 (− minus_b_over_2a root_part_over_2a))
))
```

This example uses LIST to create the list of the two values that make up the result.

Tail Recursion in Scheme:

function is tail recursive if its recursive call is the last operation in the function. For example, the member function is tail recursive.

Downloaded From EnggTree.com

```
(DEFINE (member atm a_list)
 (COND
 ((NULL? a_list) #F)
 ((EQ? atm (CAR a_list)) #T)
 (ELSE (member atm (CDR a_list)))
))
```

Functional Forms:

This section describes two common mathematical functional forms that are provided by Scheme: composition and apply-to- all.

Functional Composition:

function composition is a functional form that takes two functions as parameters and returns a function that first applies the second parameter function to its parameter and then applies the first parameter function to the return value of the second parameter function. In other words,
the function h is the composition function of f and g if h(x) = f(g(x)). For example, consider the following example:

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
```

Now the functional composition of f and g can be written as follows:

```
(DEFINE (h x) (+ 2 (* 3 x)))
```

In Scheme, the functional composition function compose can be written
as follows:

```
(DEFINE (compose f g) (LAMBDA (x)(f (g x))))
```

An Apply-to-All Functional Form:

The most common functional forms provided in functional programming languages are variations of mathematical apply- to- all functional forms. The simplest of these is map, which has two parameters: a function and a list. Map applies the given function to each element of the given list and returns a list of
the results of these applications. A Scheme definition of map follows:

```
(DEFINE (map fun a_list)
 (COND
 ((NULL? a_list) '())
 (ELSE (CONS (fun (CAR a_list)) (map fun (CDR a_list))))
))
```

Note the simple form of map, which expresses a complex functional form.As an example of the use of map, suppose we want all of the elements of a list cubed. We can accomplish this with the following:

Downloaded From EnggTree.com

EnggTree.com

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

This call returns (27 64 8 216).

Programming with ML:

ML is a static-scoped functional programming language, like Scheme.It differs from Lisp and its dialects, including Scheme, in a number of significant ways. One important difference is that ML is a strongly typed language, whereas Scheme is essentially typeless. ML has type declarations for function parameters and the return types of functions, although because of its type inferencing they are often not used. The type of every variable and expression can be statically determined. ML, like other functional programming languages, does not have variables in the sense of the imperative languages. It does have identifiers, which have the appearance of names of variables in imperative languages. However, these identifiers are best thought of as names for values.

A table called the evaluation environment stores the names of all implicitly and explicitly declared identifiers in a program, along with their types. This is like a run-time symbol table. When an identifier is declared, either implicitly or explicitly, it is placed in the evaluation environment.

Another important difference between Scheme and ML is that ML uses a syntax that is more closely related to that of an imperative language than that of Lisp. For example, arithmetic expressions are written in ML using infix notation.

Function declarations in ML appear in the general form

```
fun function_name(formal parameters) = expression;
```

Consider the following ML function declaration:

```
fun circumf(r) = 3.14159 * r * r;
```

This specifies a function named circumf that takes a floating-point (real in ML) parameter and produces a floating- point result. The types are inferred from the type of the literal in the expression. Likewise, in the function

```
fun times10(x) = 10 * x;
```

the parameter and functional value are inferred to be of type int.

Consider the following ML function:

```
fun square(x) = x * x;
```

ML determines the type of both the parameter and the return value from the *operator in the function definition. Because this is an arithmetic operator, the type of the parameter and the function are assumed to be numeric. In ML, the default numeric type is int. So, it is inferred that the type of the parameter and the return value of square is int.

If square were called with a floating-point value, as in

```
square(2.75);
```

it would cause an error, because ML does not coerce real values to int type. If we wanted square to accept real parameters, it could be rewritten as

```
fun square(x) : real = x * x;
```

Downloaded From EnggTree.com

ML does not allow overloaded functions. Each of the following definitions is also legal:

```
fun square(x : real) = x * x;
fun square(x) = (x : real) * x;
fun square(x) = x * (x : real);
```

The ML selection control flow construct is similar to that of the imperative languages. It has the following general form:

if expression then then_expression else else_expression

The first expression must evaluate to a Boolean value.

For example,

without using this pattern matching, a function to compute factorial could be written as follows:

```
fun fact(n : int): int = if n <= 1 then 1
else n * fact(n − 1);
```

Multiple definitions of a function can be written using parameter pattern matching. The different function definitions that depend on the form of the parameter are separated by an OR symbol (|).

For example, using pattern matching, the factorial function could be written as follows:

```
fun fact(0) = 1
| fact(1) = 1
| fact(n : int): int = n * fact(n − 1);
```

If fact is called with the actual parameter 0, the first definition is used; if the actual parameter is 1, the second definition is used; if an int value that is neither 0 nor 1 is sent, the third definition is used.

In ML, names are bound to values with value declaration statements of the form

val new_name = expression;

For example,

val distance = time * speed;

The val statement binds a name to a value, but the name cannot be later rebound to a new value. Actually, if you do rebind a name with a second val statement, it causes a new entry in the evaluation environment that is not related to the previous version of the name. In fact, after the new binding, the old evaluation environment entry (for the previous binding) is no longer visible. Also, the type of the new binding need not be the same as that of the previous binding. val statements do not have side effects. They simply add a name to the current evaluation environment and bind it to a value.

The normal use of val is in a let expression.

Consider the following example:

```
let val radius = 2.7
 val pi = 3.14159
```

Downloaded From EnggTree.com

```
int pi * radius * radius
end;
```

these are a filtering function for lists, filter, which takes a predicate function as its parameter. The predicate function is often given as a lambda expression, which in ML is defined exactly like a function, except with the fn reserved word, instead of fun, and of course the lambda expression is nameless. filter returns a function that takes a list as a parameter. It tests each element of the list with the predicate.

Each element on which the predicate returns true is added to a new list, which is the return value of the function. Consider the following use of filter:

```
filter(fn(x) => x < 100, [25, 1, 50, 711, 100, 150, 27, 161, 3]);
```

This application would return
[25, 1, 50, 27, 3].
The map function takes a single parameter, which is a function. The resulting function takes a list as a parameter. It applies its function to each element of the list and returns a list of the results of those applications. Consider the following code:

```
fun cube x = x * x * x;
val cubeList = map cube;
val newList = cubeList [1, 3, 5];
```

After execution, the value of newList is [1, 27, 125]. This could be done more simply by defining the cube function as a lambda expression, as in the following:

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

ML has a binary operator for composing two functions, o (a lowercase “oh”). For example, to build a function h that first applies function f and then applies function g to the returned value from f, we could use the following:

```
val h = g o f;
```

The process of currying replaces a function with more than one parameter with a function with one parameter that returns a function that takes the other parameters of the initial function.ML functions that take more than one parameter can be defined in curried form by leaving out the commas between the parameters (and the delimiting parentheses). For example, we could have the following:

```
fun add a b = a + b;
```

Although this appears to define a function with two parameters, it actually defines one with just one parameter. The add function takes an integer parameter (a) and returns a function that also takes an integer parameter (b). A call to this function also excludes the commas between the parameters, as in the following:

```
add 3 5;
```

This call to add returns 8, as expected.

Downloaded From EnggTree.com

Curried functions are interesting and useful because new functions can be constructed from them by partial evaluation. Partial evaluation means that the function is evaluated with actual parameters for one or more of the leftmost formal parameters. For example, we could define a new function as follows:

fun add5 x = add 5 x;

The add5 function takes the actual parameter 5 and evaluates the add function with 5 as the value of its first formal parameter. It returns a function that adds 5 to its single parameter, as in the following:

val num = add5 10;

The value of num is now 15.

We could create any number of new functions from the curried function add to add any specific number to a given parameter.

INTRODUCTION TO LOGIC AND LOGIC PROGRAMMING:

Languages used for logic programming are called declarative languages, because programs written in them consist of declarations rather than assignments and control flow statements. One of the essential characteristics of logic programming languages is their semantics, which is called declarative semantics. The basic concept of this semantics is that there is a simple way to determine the meaning of each statement, and it does not depend on how the statement might be used to solve a problem. Declarative semantics is considerably simpler than the semantics of the imperative languages. For example, the meaning of a given proposition in a logic programming language can be concisely determined from the statement itself.

Programming in both imperative and functional languages is primarily procedural, which means that the programmer knows what is to be accomplished by a program and instructs the computer on exactly how the computation is to be done. In other words, the computer is treated as a simple device that obeys orders.

Programming in a logic programming language is nonprocedural. Programs in such languages do not state exactly how a result is to be computed but rather describe the form of the result. The difference is that we assume the computer system can somehow determine how the result is to be computed. What is needed to provide this capability for logic programming languages is a concise means of supplying the computer with both the relevant information and a method of inference for computing desired results. Predicate calculus supplies the basic form of communication to the computer.

Sorting is commonly used to illustrate the difference between procedural and nonprocedural systems. In a language like Java, sorting is done by explaining in a Java program all of the details of some sorting algorithm to a computer that has a Java compiler. The computer, after translating the Java program into machine code or some interpretive intermediate code, follows the instructions and produces the sorted list. In a nonprocedural language, it is necessary only to describe the characteristics of the sorted list: It is some permutation of the given list such that for each pair of adjacent elements, a given relationship holds between the two elements. To

Downloaded From EnggTree.com

state this formally, suppose the list to be sorted is in an array named list that has a subscript range 1 . . . n. The concept of sorting the elements of the given list, named old_list, and placing them in a separate array, named new_list, can then be expressed as follows:
sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀j such that 11 ≤j < n, list(j)≤ list(j + 1)
where permute is a predicate that returns true if its second parameter array is a permutation of its first parameter array.

PROGRAMMING WITH PROLOG:

Terms: Prolog programs consist of collections of statements. There are only a few kinds of statements in Prolog, but they can be complex. All Prolog statement, as well as Prolog data, are constructed from terms.
A Prolog term is a constant, a variable, or a structure. A constant is either an atom or an integer. An atom is either a string of letters, digits, and underscores that begins with a lowercase letter or a string of any printable ASCII characters delimited by apostrophes.

A variable is any string of letters, digits, and underscores that begins with an uppercase letter or an underscore (_). Variables are not bound to types by declarations. The binding of a value, and thus a type, to a variable is called an instantiation. A variable that has not been assigned a value is called uninstantiated.

Fact Statements:

Prolog has two basic statement forms; these correspond to the headless and headed Horn clauses of predicate calculus. The simplest form of headless Horn clause in Prolog is a single structure, which is interpreted as an unconditional assertion, or fact. Logically, facts are simply propositions that are assumed to be true. The following examples illustrate the kinds of facts one can have in a Prolog program. Notice that every Prolog statement is terminated by a period.

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

These simple structures state certain facts about jake, shelley, bill, and mary. For example, the first states that shelley is a female. The most common and straightforward meaning is bill is the father of jake.

Rule Statement:

This form can be related to a known theorem in mathematics from which a conclusion can be drawn if the set of given conditions is satisfied. The right side is the antecedent, or if part,

Downloaded From EnggTree.com

and the left side is the consequent, or then part. If the antecedent of a Prolog statement is true, then the consequent of the statement must also be true.

Conjunctions contain multiple terms that are separated by logical AND operations. In Prolog, the AND operation is implied.

The general form of the Prolog headed Horn clause statement is

consequence :- antecedent_expression.

It is read as follows: “consequence can be concluded if the antecedent expression is true or can be made to be true by some instantiation of its variables.”

For example,

ancestor(mary, shelley) :- mother(mary, shelley).

states that if mary is the mother of shelley, then mary is an ancestor of shelley.

Goal Statements:

These statements are the basis for the theorem-proving model. The theorem is in the form of a proposition that we want the system to either prove or disprove. In Prolog, these propositions are called goals, or queries. For example, we could have

man(fred).

to which the system will respond either yes or no. The answer yes means that the system has proved the goal was true. The answer no means that either the goal was determined to be false or the system was simply unable to prove it.

The Inferencing Process of Prolog:

Queries are called goals. When a goal is a compound proposition, each of the facts (structures) is called a subgoal. To prove that a goal is true, the inferencing process must find a chain of inference rules and/or facts in the database that connect the goal to one or more facts in the database. For example, if Q is the goal, then either Q must be found as a fact in the database or the inferencing process must find a fact P1 and a sequence of propositions P2, P3, c , Pn such that

P2 :- P1

P3 :- P2

. . .

Q :- Pn

Because the process of proving a subgoal is done through a proposition-matching process, it is sometimes called matching. In some cases, proving a subgoal is called satisfying that subgoal. Consider the following query:

man(bob).

father(bob).

man(X) :- father(X).

Prolog would be required to find these two statements and use them to infer the truth of the goal. There are two opposite approaches to attempting to match a given goal to a fact in the database. The system can begin with the facts and rules of the database and attempt to find a sequence of

Downloaded From EnggTree.com

EnggTree.com

matches that lead to the goal. This approach is called bottom-up resolution, or forward chaining. The alternative is to begin with the goal and attempt to find a sequence of matching propositions that lead to some set of original facts in the database. This approach is called top-down resolution, or backward chaining.

The following example illustrates the difference between forward and backward chaining. Consider the query:

man(bob).

Assume the database contains

father(bob).

man(X) :- father(X).

Forward chaining would search for and find the first proposition. The goal is then inferred by matching the first proposition with the right side of the second rule (father(X)) through instantiation of X to bob and then matching the left side of the second proposition to the goal. Backward chaining would first match the goal with the left side of the second proposition (man(X)) through the instantiation of X to bob. As its last step, it would match the right side of the second proposition (now father(bob)) with the first proposition.

The next design question arises whenever the goal has more than one structure, as in our example. The question then is whether the solution search is done depth first or breadth first. A depth-first search finds a complete sequence of propositions—a proof—for the first subgoal before working on the others. A breadth-first search works on all subgoals of a given goal in parallel.

When a goal with multiple subgoals is being processed and the system fails to show the truth of one of the subgoals, the system abandons the subgoal it cannot prove. It then reconsiders the previous subgoal, if there is one, and attempts to find an alternative solution to it. This backing up in the goal to the reconsideration of a previously proven subgoal is called backtracking. A new solution is found by beginning the search where the previous search for that subgoal stopped.

Simple Arithmetic:

Prolog supports integer variables and integer arithmetic. Originally, the arithmetic operators were functors, so that the sum of 7 and the variable X was formed with +(7, X) Prolog now allows a more abbreviated syntax for arithmetic with the is operator. This operator takes an arithmetic expression as its right operand and a variable as its left operand. All variables in the expression must already be instantiated, but the left-side variable cannot be previously instantiated. For example, in A is B / 17 + C. if B and C are instantiated but A is not, then this clause will cause A to be instantiated with the value of the expression.

This basic information can be coded as facts, and the relationship between speed, time, and distance can be written as a rule, as in the following:

speed(ford, 100).

Downloaded From EnggTree.com

EnggTree.com

```
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed),time(X, Time),
 Y is Speed * Time.
```

Now, queries can request the distance traveled by a particular car. For example, the query

```
distance(chevy, Chevy_Distance).
```

instantiates Chevy_Distance with the value 2205.

MULTI-PARADIGM LANGUAGES:

A programming language that supports both procedural and object-oriented programming concepts. C++ is one of the multi paradigm language because it has the concepts of programming language but added the object oriented concepts also.

Downloaded From EnggTree.com