LECTURE NOTES ON

PRINCIPLES OF PROGRAMMING LANGUAGES (15A05504)

III B.TECH I SEMESTER

(JNTUA-R15)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA

Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112

(Approved by AICTE, New Delhi Affiliated to JNTUA Ananthapuramu. ISO 9001:2015 Certified Institute)

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR
B. Tech III-I Sem. (CSE)

L T P C

3 10 3

15A05504 PRINCIPLES OF PROGRAMMING LANGUAGES

Unit I:
Introduction: Software Development Process, Language and Software DevelopmentEnvironments, Language and Software Design Models, Language and ComputerArchitecture, Programming Language Qualities, A brief Historical Perspective.
Syntax and Semantics: Language Definition, Language Processing, Variables,Routines, Aliasing and Overloading, Run-time Structure.

Unit II:
Structuring the data: Built-in types and primitive types, Data aggregates and typeconstructors, User-defined types and abstract data types, Type Systems, The typeStructure of representative languages, Implementation Models

Unit III:
Structuring the Computation: Expressions and Statements, Conditional Executionand Iteration, Routines, Exceptions, Pattern Matching, Nondeterminism andBacktracking, Event-driven computations, Concurrent Computations
Structuring the Program: Software Design Methods, Concepts in Support ofModularity, Language Features for Programming in the Large, Generic Units

Unit IV:
Object-Oriented Languages: Concepts of Object-oriented Programming, Inheritancesand the type system, Object-oriented features in programming languages

Unit V:
Functional Programming Languages: Characteristics of imperative languages,Mathematical and programming functions, Principles of Functional Programming,Representative Functional Languages, Functional Programming in C++
Logic and Rule-based Languages: ―What‖ versus ―how‖: Specification versusimplementation, Principles of Logic Programming, PROLOG, Functional Programmingversus Logic Programming, Rule-based Languages

Textbook:
1) ―Programming Language Concepts‖, Carlo Ghezzi, Mehdi Jazayeri, WILEY Publications. Third Edition, 2014

Reference Textbooks:
1. Concepts of Programming Languages, Tenth Edition, Robert W. Sebesta, PearsonEducation.
2. Programming Languages Principles and Paradigms, Second Edition, Allen B. Tucker,Robert E. Noonan, McGraw Hill Education.
3. Introduction to Programming Languages, Aravind Kumar Bansal, CRC Press.

UNIT 1

INTRODUCTION

Software Development Process:

The software is said to have a *life cycle* com-posed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a fragment of specification, a test plan or a users manual. In the traditional *waterfall model* of the software life cycle.

A sample software development process based on the waterfall model may be comprised of the following phases:

Requirement analysis and specification**:** The purpose of this phase is to identify anddocument the exact requirements for the system.These requirements are developed jointly by users and software developers. The result of this phase is a requirements document stating what the system should do, along with users' manuals, feasibility and cost studies, performance requirements, and so on. The requirements document does not specify how the system is going to meet its requirements.

Software design and specification**:** Starting with the requirements document, softwaredesigners design the software system. The result of this phase is a system design specification document identifying all of the modules comprising the system and their interfaces.

***Implementation* (coding)**: The system is implemented to meet the design specified in theprevious phase. The design specification, in this case, states the " what" ; the goal of the implementation step is to choose how, among the many possible ways, the system shall be coded to meet the design specification. The result is a fully implemented and documented system.

Verification and validation: This phase assesses the quality of the implemented system,which is then delivered to the user. Note that this phase should not be concentrated at the end of the implementation step, but should occur in every phase of software development to check that intermediate deliverables of the process satisfy their objectives. The checks are accomplished by answering the following two questions: "Are we building the product right?" "Are we building the right product?" Two specific kinds of assessment performed during implementation are *module testing* and *integration testing*.

Maintenance**:** Following delivery of the system, changes to the system may becomenecessary either because of detected malfunctions, or a desire to add new capabilities or to improve old ones, or changes that occurred in operational environment.

Programming languages are used only in some phases of the development process. They are obviously used in the implementation phase, when algorithms and data structures are defined and coded for the modules that form the entire application. Moreover, modern higher-level languages are also used in the design phase, to describe precisely the decomposition of the entire application into modules, and the relationships among modules, before any detailed implementation takes place.

Language and Software Development Environments:

A *software developmentenvironment* we mean an integrated set of tools and techniques that aids in thedevelopment of software. The environment is used in all phases of software development: requirements, design, implementation, verification and validation, and maintenance.

The work in any of the phases of software development may be supported by computer-aided tools. The phase currently supported best is the coding phase, with such tools as *text editors, compilers, linkers,* and *libraries*. These tools have evolved gradually, as the need for automation has been recognized. Nowadays, one can normally use an interactive editor to create a program and the file system to store it for future use. When needed, several previously created and (possibly) compiled programs may be linked to produce an executable program. A debugger is commonly used to locate faults in a program and eliminate them. These computer-aided program development tools have increased programming productivity by reducing the chances of errors.

Language and Software Design Models:

The relationship between software design methods and programming languages is an important one. To understand the relationship between a programming language and a design method, it is important to realize that programming languages may enforce a certain programming style, often called a *programming paradigm*.

Here we review the most prominent programming language paradigms, with special emphasis on the unit of modularization promoted by the paradigm.

Procedural programming:This is the conventional programming style,where programs are decomposed into computation steps that perform complex operations. Procedures and functions (collectively called routines) are used as modularization units to define such computation steps.

Functional programming:The functional style of programming is rooted inthe theory of mathematical functions. It emphasizes the use of expressions and functions. The functions are the primary building blocks of the program; they may be passed freely as parameters and may be constructed and returned as result parameters of other functions.

Abstract data type programming**:** Abstract-data type (ADT) programmingrecognizes abstract data types as the unit of program modularity. CLU was the first language designed specifically to support this style of programming.

Module-based programming**:** Rather than emphasizing abstract-data types,module-based programming emphasizes modularization units that are groupings of entities such as variables, procedures, functions, types, etc. A program is composed of a set of such modules. Modula-2 and Ada support this style of programming.

Object-oriented programming**:** The object-oriented programming style emphasizes the definition of classes of objects. Instances of classes are created by the program as needed during program execution. This style is based on the definition of hierarchies of classes and run-time selection of units to execute. Smalltalk and Eiffel are representative languages of this class. C++ and Ada 95 also support the paradigm.

Generic programming: This style emphasize the definition of generic modules that may be instantiated, either at compile-time or runtime, to create the entities data structures, functions, and procedures needed to form the program. This approach to programming encourages the development of high-level, generic, abstractions as units of modularity. It can exist jointly with object-oriented programming, as in Eiffel, with functional programming, as in ML. It also exists in languages that provide more than one paradigm, like Ada and C++.

***Declarative programming*:** This style emphasizes the declarative description of a problem, rather than the decomposition of the problem into an algorithmic implementation. As such, programs are close to a specification. Logic languages, like PROLOG, and rule-based languages, like OPS5 and KEE, are representative of this class of languages.

Language and Computer Architecture:

Languages have been constrained by the ideas of Von Neumann, because most current computers are similar to the original Von Neumann architecture

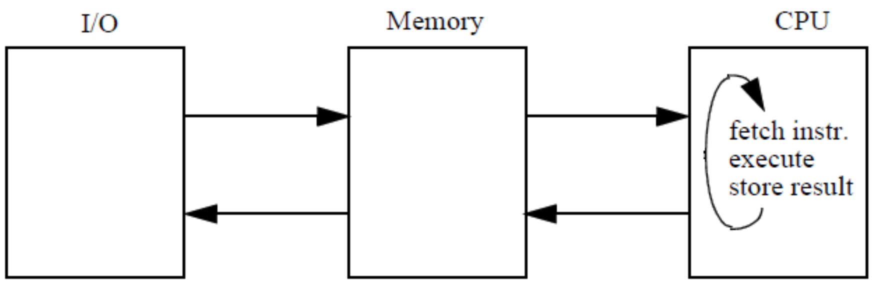


Figure 1.A Von Neumann computer architecture

The Von Neumann architecture, sketched in Figure 1, is based on the idea of a memory that contains data and instructions, a CPU, and an I/O unit. The CPU is responsible for taking instructions out of memory, one at a time. Machine instructions are very low-level. They require the data to be taken out of memory, manipulated via arithmetic or logic operations in the CPU, and the results copied back to some memory cells. Thus, as an instruction is executed, the *state* of the machine changes.

Conventional programming languages can be viewed as abstractions of an underlying Von Neumann architecture. For this reason, they are called Von Neumann languages. An *abstraction* of a phenomenon is a model which ignores irrelevant details and highlights the relevant aspects. Conventional languages based on the Von Neumann computation model are often called *imperative languages*. Other common terms are *state-based languages*, or *statement-based languages*, or simply *Von Neumann languages*.

The historical developments of imperative languages have gone through increasingly higher levels of abstractions. In the early times of computing,.

Many kinds of abstractions were later invented by language designers, such as procedures and functions, data types, exception handlers, classes, concurrency features, etc. As suggested by Figure 2, language developers tried to make the level of programming languages higher, to make languages easier to use by humans, but still based the concepts of the language on those of the underlying Von Neumann architecture.

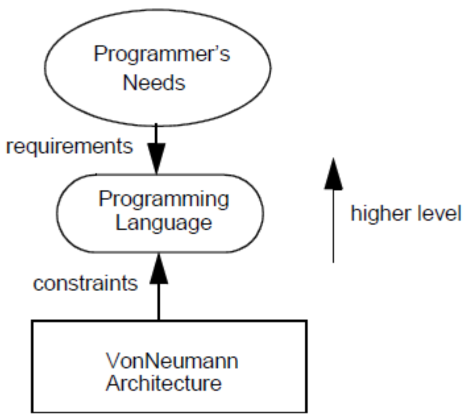


Figure 2. Requirements and constraints on a language

Other kinds of parallel languages exist supporting parallelism at a much finer granularity, which do not fall under the classification shown in Figure 3.

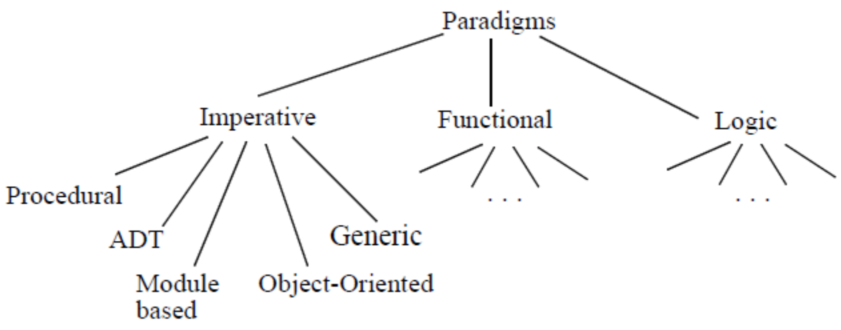


Figure 3. Hierarchy of paradigms

<u>Programming Language Qualities:</u>

A programming language is a tool for the development of software. Thus, ultimately, the quality of the language must be related to the quality of the software.

Software must be reliable. Users should be able to rely on the software, i.e.,the chance of failures due to faults in the program should be low.

The reliability goal is promoted by several programming language qualities.

- *Writability*. It refers to the possibility of expressing a program in a way thatis natural for the problem. The programmer should not be distracted by details and tricks of the language from the more important activity of problem solving. For example, an assembly language programmer is often distracted by the addressing mechanisms needed to access certain data, such as the positioning of index registers, and so on. The easier it is to concentrate on the problem-solving activity, the less error-prone is program writing and the higher is productivity.
- *Readability*. It should be possible to follow the logic of the program and todiscover the presence of errors by examining the program.
- *Simplicity*. A simple language is easy to master and allows algorithms to beexpressed easily, in a way that makes the programmer self-confident. Simplicity can obviously conflict with power of the language. For example, Pascal is simpler, but less powerful than

- *Safety*. The language should not provide features that make it possible towrite harmful programs.For example, a language that does not provide **goto** statements nor pointer variables eliminates two well-known sources of danger in a program. Such features may cause subtle errors that are difficult to track during program development,
- *Robustness*. The language supports robustness whenever it provides theability to deal with undesired events (arithmetic overflows, invalid input, and so on). That is, such events can be trapped and a suitable response can be programmed to respond to their occurrence.

Software must be maintainable. Existing software must be modified to meet new requirements. Also, because it is almost impossible to get the real requirements right in the first place, for such complex systems one can only hope to gradually evolve a system into the desired one.

Two main features that languages can provide to support modification are factoring and locality.

- *Factoring*. This means that the language should allow programmers to fac-tor related features into one single point. As a very simple example, if an identical operation is repeated in several points of the program, it should be possible to factor it in a routine and replace it by a routine call.
- *Locality*. This means that the effect of a language feature is restricted to asmall, local portion of the entire program. Otherwise, if it extends to most of the program, the task of making the change can be exceedingly complex. For example, in abstract data type program-ming, the change to a data structure defined inside a class is guaranteed not affect the rest of the program as long as the opera-tions that manipulate the data structure are invoked in the same

Software must execute efficiently. This goal affects both the programming language (features that can be efficiently implemented on present-day architectures) and the choice of algorithms to be used.

The need for efficiency has guided language design from the beginning. Many languages have had efficiency as a main design goal, either implicitly or explicitly. For example, FORTRAN originally was designed for a specific machine (the IBM 704). Many of FORTRAN's restrictions, such as the number of array dimensions or the form of expressions used as array indices, were based directly on what could be implemented efficiently on the IBM 704.

Efficiency is often a combined quality of both the language and its implementation. The language adversely affects efficiency if it disallows certain optimizations to be applied by the compiler. The implementation adversely affects efficiency if it does not take all opportunities into account in order to save space and improve speed.

A brief Historical Perspective:

Table 1: Genealogy of selected programming languages

Language	Year	Originator	Predecessor Language	IntendedPurpose	Reference
FORTRAN	1954-57	J. Backus		Numericcomputing	Glossary

ALGOL 60	1958-60	Committee	FORTRAN	Numericcomputing	Naur 1963
COBOL	1959-60	Committee		Businessdata processing	Glossary
APL	1956-60	K. Iverson		Array processing	Iverson 1962
LISP	1956-62	J. McCarthy		Symboliccomputing	Glossary
SNOBOL4	1962-66	R. Griswold		String processing	Griswold et al.
PL/I	1963-64	Committee	FORTRAN ALGOL 60 COBOL	Generalpurpose	ANSI 1976
SIMULA 67	1967	O.-J.Dahl	ALGOL 60	Simulation	Birtwistle et al.1973
ALGOL 68	1963-68	Committee	ALGOL 60	General purpose	vanWijngaardenet al. 1976 Lindsay and vanderMeulen 1977
Pascal	1971	N. Wirth	ALGOL 60	Educationaland gen. purpose	Glossary
PROLOG	1972	A. Colmerauer		Artificialintelligence	Glossary
C	1974	D. Ritchie	ALGOL 68	Systemsprogramming	Glossary
Mesa	1974	Committee	SIMULA 67	Systems programming	Geschke et al.1977
SETL	1974	J. Schwartz		Very highlevel lang.	Schwartz et al.1986
Concurrent Pascal	1975	P. Brinch Hansen	Pascal	Concurrent programming	Brinch Hansen1977
Scheme	1975	Steele andSussman (MIT)	LISP	Educationusing functional programming	Abelson and Sussman 1985
CLU	1974-77	B. Liskov	SIMULA 67	ADTprogramMing	Liskov et al. 1981
Euclid	1977	Committee	Pascal	Verifiableprograms	Lampson et al.1977
Gypsy	1977	D. Good	Pascal	Verifiableprograms	Ambler et al.1977
Modula-2	1977	N. Wirth	Pascal	Systems programming	Glossary
Ada	1979	J. Ichbiah	Pascal SIMULA 67	Generalpurpose Embeddedsystems	Glossary
Smalltalk	1971-80	A. Kay	SIMULA 67 LISP	Personalcomputing	Glossary
C++	1984	B. Strous-trup	C SIMULA 67	Generalpurpose	Glossary

Syntax and Semantics

Language Definition:

A language definition should enable a person or a computer program to determine (1) whether a purported program is in fact valid, and (2) if the program is valid, what its meaning or effect is. In general, two aspects of a language-program-ming or natural language-must be defined: syntax and semantics.

Syntax:

Syntax is described by a set of rules that define the form of a language: they define how sentences may be formed as sequences of basic constituents called words. Using these rules we can tell whether a sentence is legal or not. The syntax does not tell us anything about the content (or meaning) of the sentence– the semantic rules tell us that. As an example, C keywords (such as while, do, if, else,...), identifiers, numbers, operators, ... are words of the language. The C syntax tells us how to combine such words to construct well-formed statements and programs

The syntax of a language is defined by two sets of rules: lexical rules and syntactic rules. Lexical rules specify the set of characters that constitute the alphabet of the language and the way such characters can be combined to form valid words. For example, Pascal considers lowercase and uppercase characters to be iden-tical, but C and Ada consider them to be distinct. Thus, according to the lexi-cal rules, " Memory" and " memory" refer to the same variable in Pascal, but to distinct variables in C and Ada. The lexical rules also tell us that <> (or ¦) is a valid operator in Pascal but not in C, where the same operator is represented by !=. Ada differs from both, since " not equal" is represented as /=; delimiter <> (called " box") stands for an undefined range of an array index.

How does one define the syntax of a language? FORTRAN was defined by simply stating some rules in English. ALGOL 60 was defined with a context-free grammar developed by John Backus. This method has become known as BNF or Backus Naur form (Peter Naur was the editor of the ALGOL 60 report.) BNF provides a compact and clear definition for the syntax of programming languages.

EBNF is a meta-language. A meta-language is a language that is used to describe other languages. We describe EBNF first, and then we show how it can be used to describe the syntax of a simple programming language (Figure 4(a)). The symbols ::=, <, >, *, +, (,), and | are symbols of the metalanguage: they are *metasymbols*. A language is described in EBNF through a set of rules. For example, <program> ::= { <statement>* } is a rule. The symbol "::=" stands for " is defined as" . The symbol " *" stands for " an arbitrary sequence of the previous element" . Thus, the rule states that a <program> is defined as an arbitrary sequence of <statement> within brackets " {" a nd " }" . The entities inside the metalanguage brackets " <" , and " >" are called nonter-minals; an entity such as the " }" above is called a terminal. Terminals are what we have previously called words of the language being defined, whereas nonterminals are linguistic entities that are defined by other EBNF rules. In order to distinguish between metasymbols and terminals, Figure 5 uses the convention that terminals are written in bold. To complete our description of EBNF, the metasymbol" +" denotes one or more repetitions of the previous element (i.e., at least one element must be present, as opposed to " *"). The metasymbol" |" denotes a choice. For example, a <statement> is described in Figure 5(a) as either an<assignment>, a <conditional>, or a <loop>.

(a) Syntax rules

<program>::=**{**<statement>* **}**
<statement>::=<assignment> | <conditional> | <loop>
<assignment>::=<identifier>**=**<expr> ;
<conditional>::=**if**<expr>**{**<statement> + **}** |
if<expr> **{** <statement> + **} else {** <statement>
+ **}** <loop>::=**while**<expr>**{**<statement> + **}**
<expr> ::=<identifier> | <number> | (<expr>) |
<expr><operator><expr>

(b) Lexical rules

<operator>::= **+** | **-** | ***** | **/** | **=** | **¦**| **<** | **>** | **£** | **³**

<identifier>::= <letter><ld>*
<ld>::= <letter> | <digit>
<number>::= <digit>+
<letter>::= **a** | **b** | **c** | . . . | **z**

FIGURE 4.EBNF definition of a simple programming language (a) syntax rules, (b) lexical rules

The lexical rules, which describe how identifiers, numbers, and operators look like in our simple language are also described in EBNF, and shown in Figure 4(b). To do so, <operator>, <identifier>, and <number>, which are words of the language being defined, are detailed in terms of elementary symbols of the alphabet.

We illustrate an extended version of BNF (EBNF) bellow, along with the definition of a simple language. Syntax diagrams provide another way of defining syntax of programming languages. They are conceptually equivalent to BNF, but their pictorial notation is somewhat more intuitive. Syntax diagrams are also described below.
Figure 5 shows the syntax diagrams for the simple programming language whose EBNF has been discussed above. Nonterminals are represented by cir-cles and terminals by boxes. The nonterminal symbol is defined with a transi-tion diagram having one entry and one exit edge.

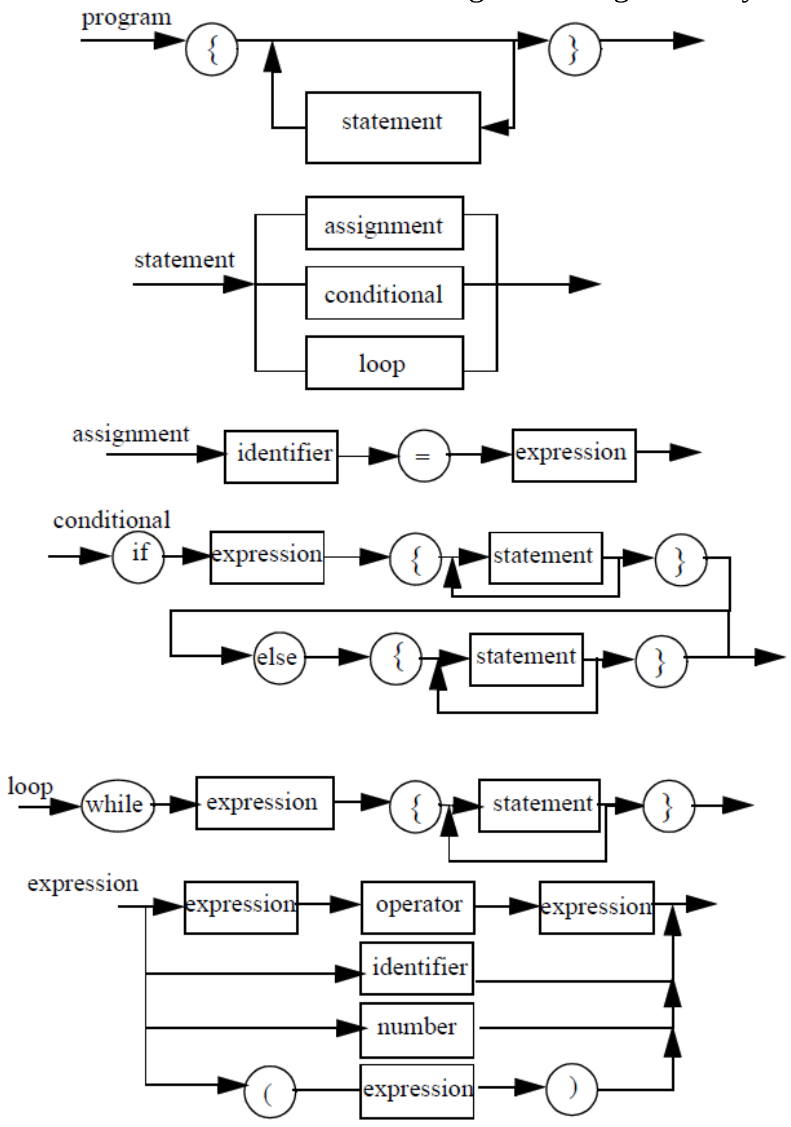


FIGURE 5. Syntax diagrams for the language described in Figure 4 .

Semantics:

Syntax defines well-formed programs of a language. Semantics defines the meaning of syntactically correct programs in that language. For example, the semantics of C help us determine that the declaration

```
int vector [10];
```

causes ten integer elements to be reserved for a variable named vector. The first element of the vector may be referenced by vector [0]; all other elements may be referenced by an index i, $0 \leq i \leq 9$.

As another example, the semantics of C states that the instruction

```
if (a > b) max = a; else max = b;
```

means that the expression a > b must be evaluated, and depending on its value, one of the two given assignment statements is executed. Note that the syntax rules tell us how to form this statement– for example, where to put a " ;" – and the semantic rules tell us what the effect of the statement is.

While syntax diagrams and BNF have become standard tools for syntax description, no such tool has become widely accepted and standard for semantic description. Different formal approaches to semantic definition exist, but none is entirely satisfactory.

There are two ways of formally specifying semantics: axiomatic semantics and denotational semantics

Axiomatic semantics views a program as a state machine. Programming lan-guage constructs are described by describing how their execution causes a state change. A state is described by a first-order logic predicate which defines the property of the values of program variables in that state. Thus the meaning of each construct is defined by a rule that relates the two states before and after the execution of that construct.

A predicate P that is required to hold after execution of a statement S is called a *postcondition* for S. A predicate Q such that the execution of S terminates and postcondition P holds upon termination is called a *precondition* for S and p.

Axiomatic semantics specifies each statement of a language in terms of a function asem, called the *predicate transformer,* which yields the weakest pre-condition W for any statement S and any postcondition P. It also provides composition rules that allows the precondition to be evaluated for a given program and a given postcondition. Let us consider an assignment statement

```
x = expr;
```

and a postcondition P. The weakest precondition is obtained by replacing each occurrence of x in P with expression expr.

$$\text{asem } (x = \text{expr};, P) = P_{x \rightarrow \text{expr}}$$

The specification of semantics of selection is straightforward. If B is a bool-ean expression and L1, L2 are two statement lists, then let **if**-stat be the follow-ing statement:

ifB **then** L1 **else** L2

If P is the postcondition that must be established by **if**-stat, then the weakest precondition is given by

asem (**if**-stat, P) = (B Éasem (L1, P)) and (not B Éasem (L2, P))

That is, function asem yields the semantics of either branch of the selection, depending on the value of the condition.

Denotational semantics associates each language statement with a functiondsem from the state of the program before the execution to the state after exe-cution. The *state* (i.e., the values stored in the memory) is represented by a function mem from the set of program identifiers ID to values. Thus denota-tional semantics differs from axiomatic semantics in the way states are described (functions vs. predicates). For simplicity, we assume that values can only belong to type integer.

Let us start our analysis from arithmetic expressions and assignments. For an expression expr, mem (expr) is defined as error if mem (v) is undefined for some variable v occurring in expr. Otherwise mem (expr) is the result of evaluating expr after replacing each variable v in expr with mem (v).

If x = expr is an assignment statement and mem is the function describing the memory before executing the assignment

dsem (x := expr, mem) = error

if mem (x) is undefined for some variable x occurring in expr. Otherwise

dsem (x: = expr, mem) = mem'

where mem' (y) = mem (y) for all y ¦x, mem' (x) = mem (expr).

As axiomatic semantics, denotational semantics is defined compositionally. That is, given the state transformation caused by each individual statement, it provides the state transformation caused by compound statements and, eventually, by the entire program.

<u>Language Processing</u>:

Machine languages are designed on the basis of speed of execution, cost of realization, and flexibility in building new software layers upon them. On the other hand, programming languages often are designed on the basis of the ease and reliability of programming. A basic problem, then, is how a higher-level language eventually can be executed on a computer whose machine language is very different and at a much lower level.

There are generally two extreme alternatives for an implementation: interpretation and translation.

Interpretation

In this solution, the actions implied by the constructs of the language are executed directly (see Figure 6). Usually, for each possible action there exists a subprogram– written in machine language– to execute the action. Thus, interpretation of a program is accomplished by calling subprograms in the appropriate sequence.

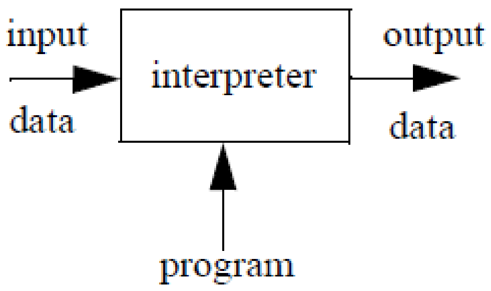


FIGURE 6. Language processing by interpretation

More precisely, an interpreter is a program that repeatedly executes the following sequence.

```
Get the next statement;
Determine the actions to be executed;
Perform the actions;
```

Translation

In this solution, programs written in a high-level language are translated into an equivalent machine-language version before being executed. This translation is often performed in several steps (see Figure 7). Program modules might first be separately translated into relocatable machine code; modules of relocatable code are linked together into a single relocatable unit; finally, the entire program is loaded into the computer's memory as executable machine code. The translators used in each of these steps have specialized names: compiler, linker (or linkage editor), and loader, respectively.

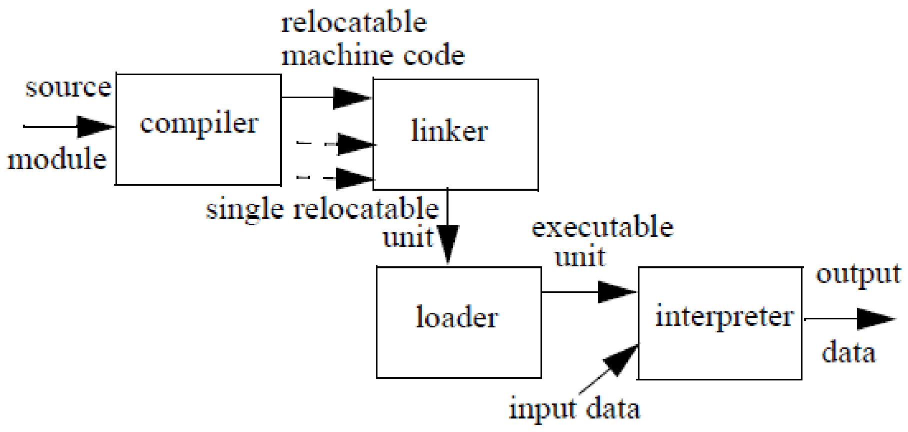


FIGURE 7. Language processing by Translation

Compilers and interpreters differ in the way they can report on run-time errors. Typically, with compilation, any reference to the source code is lost in the generated object code. If an error is generated at run-time, it may be impossible to relate it to the source language construct being executed. This is why run-time error messages are often obscure and almost meaningless to the programmer. On the opposite, the interpreter processes source statements, and can relate a run-time error to the source statement being executed. For these reasons, certain programming environments contain both an interpreter and a compiler for a given programming language. The interpreter is used while the program is being developed, because of its improved diagnostic facilities. The compiler is then used to generate efficient code, after the pro-gram has been fully validated.

Variables:

Formally, a variable is a 5-tuple <name, scope, type, l_value, r_value>, where

- name is a string of characters used by program statements to denote the variable;
- scope is the range of program instructions over which the name is known;
- type is the variable’s type;
- l_value is the memory location associated with the variable;
- r_value is the encoded value stored in the variable’s location.

These attributes are described below,

Name and scope

A variable’s *name* is usually introduced by a special statement, called *declaration* and, normally, the variable’s*scope*extends from that point until somelater closing point, specified by the language. The scope of a variable is the range of program instructions over which the name is known.

For example, consider the following example of a C program:

```
# include
<stdio.h> main (
)
{
    int x, y;
    scanf ("%d %d", &x, &y);
        /*two decimal values are read and stored in the l_values of x and y */
    {
        /*this is a block used to swap x and
        y*/ int temp;
        temp = x;
        x = y;
        y = temp;
    }
    printf ("%d %d", x, y);
}
```

The declaration int x, y; makes variables named x and y visible throughout pro-gram main. The program contains an internal block, which groups a declaration and statements. The declaration int temp; appearing in the block makes a variable named temp visible within the inner block, and invisible outside. Thus, it would be impossible to insert temp as an argument of operation printf.

Variables can be bound to a scope either statically or dynamically. *Staticscope binding* defines the scope in terms of the lexical structure of a program,that is, each reference to a variable can be statically bound to a particular (implicit or explicit) variable declaration by examining the program text, without executing it.*Dynamic scope binding* defines the scope of a variable's name in terms of pro-gram execution. Typically, each variable declaration extends its effect over all the instructions executed thereafter, until a new declaration for a variable with the same name is encountered during execution. APL, LISP (as origi-nally defined), and SNOBOL4 are examples of languages with dynamic scope rules.

Type

In this section we provide a preliminary introduction to types. The subject will be examined in more depth in Chapters 3 and 6. We define the type of a variable as a specification of the set of values that can be associated with the variable, together with the operations that can be legally used to create, access, and modify such values. A variable of a given type is said to be an *instance* of the type.

In some languages, the programmer can define new types by means of type declarations. For example, in C one can write

```
typedefint vector [10];
```

This declaration establishes a binding– at translation time–b etween the type name vector and its implementation (i.e., an array of 10 integers, each accessi-ble via an index in the subrange 0. .9). As a consequence of this binding, type vector inherits all the operations of the representation data structure (the array); thus, it is possible to read and modify each component of an object of type vector by indexing within the array.

Traditional languages, such as FORTRAN, COBOL, Pascal, C, C++, Mod-ula-2, and Ada bind variables to their type at compile time, and the binding cannot be changed during execution. This solution is called *static typing*. In these languages, the binding between a variable and its type is specified by a variable declaration. For example, in C one can write:

```
int x, y;

char c;
```

By declaring variables to belong to a given type, variables are automatically protected from the application of illegal (or nonsensical) operations. For example, in Ada the compiler can detect the application of the illegal assign-mentI:= not A, if I is declared to be an integer and A is a boolean.

l_value

The l_value of a variable is the storage area bound to the variable during exe-cution. The *lifetime*, or extent, of a variable is the period of time in which such binding exists. The storage area is used to hold the r_value of the vari-able. We will use the term *data object* (or simply, object) to denote the pair <l_value, r_value>.

The action that acquires a storage area for a variable– and thus establishes the binding– is called *memory allocation*. The lifetime extends from the point of allocation to the point in which the allocated storage is reclaimed (*memorydeallocation*). In some languages, for some kinds of variables, allocation isperformed before run time and storage is only reclaimed upon termination (*static allocation*). In other languages, it is performed at run time (*dynamicallocation*), either upon explicit request from the programmer via a creationstatement or automatically, when the variable's declaration is encountered, and reclaimed during execution.

r_value

The r_value of a variable is the encoded value stored in the location associ-ated with the variable (i.e., itsl_value). The encoded representation is inter-preted according to the variable's type. For example, a certain sequence of bits stored at a certain location would be interpreted as an integer number if the variable’s type is int; it would be interpreted as a string if the type is an array of char.

l_values and r_values are the main concepts related to program execution. Program instructions access variables through their l_value and possibly modify their r_value. The terms l_value and r_value derive from the conven-tional form of assignment statements, such as x = y; in C. The variable appear-ing at the left-hand side of the assignment denotes a location (i.e., its l_value is meant).The variable appearing at the right-hand side of the assignment denotes the contents of a location (i.e., its r_value is meant). Whenever no ambiguity arises, we use the simple term “ value” of a variable to denote its r_value.

Routines:

Programming languages allow a program to be composed of a number of units, called *routines*. Routines can be developed in a more or less independent fashion and can sometimes be translated separately and combined after translation. Assembly language subprograms, FORTRAN subroutines, Pascal and Ada procedures and functions, C functions are well-known examples of routines.

In the existing programming language world, routines usually come in two forms: procedures and functions. Functions return a value; procedures do not. Some languages, e.g., C and C++, only provide functions, but procedures are easily obtained as functions returning the null value void. Bellow program shows the example of a C function definition.

```
/* sum is a function which computes the sum
of the first n positive integers, 1 + 2 + ... + n;
parameter n is assumed to be positive */
int sum (int n)
{
    inti, s;
    s = 0;
    for (i = 1; i<= n ; ++i)
        s+= i;
    return s:
}
```

Like variables, routines have a name, scope, type, l_value, and r_value. A routine name is introduced in a program by a routine *declaration*. Usually the scope of such name extends from the declaration point on to some closing construct, statically or dynamically determined, depending on the language. For example, in C a function declaration extends the scope of the function till the end of the file in which the declaration occurs.

Some languages (like Pascal, Ada, C, and C++) distinguish between *declaration* and *definition* of a routine. A routine declaration introduces the routine’sheader, without specifying the body. The name is visible from the declaration point on, up to the scope end. The definition specifies both the header and the body. The distinction between declaration and definition is necessary to allow routines to call themselves in a mutual recursion scheme which is illustrates bellow.

```
int A (int x, int y); // declaratiuon of afunction with two int
                      // parameters and returning an int
                      // A is visible from this point on
   float B (int z) //this is a definition of a function; B is visible from this
   point on {
       int w, u;
       . . .
       w = A (z, u); //calls A, which is visible at this point
       . . .
   };
   int A (int x, int y) //this is A’s definition
   {
       float t;
```

```
        . . .
        t = B (x); //B is visible here
        . . .
    }
```

Aliasing and Overloading:

The language uses special names (denoted by operators), such as + or * to denote certain predefined operations. So far, we implicitly assumed that at each point in a pro-gram a name denotes exactly one entity, based on the scope rules of the language. Since names are used to identify the corresponding entity, the assumption of unique binding between a name and an entity would make the identification unambiguous. This restriction, however, is almost never true for existing programming languages.

For example, in C one can write the following fragment:

```
inti, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
```

In the example, operator + in the two instructions of the program denotes two different entities. In the first expression, it denotes integer addition; in the second, it denotes floating-point addition. Although the name is the same for the operator in the two expressions, the binding between the operator and the corresponding operation is different in the two cases, and the exact binding can be established at compile time, since the types of the operands allow for the disambiguation.

We can generalize the previous example by introducing the concept of *over-loading*. A name is said to be overloaded if more than one entity is bound tothe name at a given point of a program and yet the specific occurrence of the name provides enough information to allow the binding to be uniquely estab-lished. In the previous example, the types of the operands to which + is applied allows for the disambiguation.

As another example, if the second instruction of the previous fragment would be changed to

```
a = b + c + b ( );
```

the two occurrences of name b would (unambiguously) denote, respectively, variable b and routine b with no parameters and returning a float value (assum-ing that such routine is visible by the assignment instruction). Similarly, if another routine named b, with one int parameter and returning a float value is visible, instruction

```
a = b ( ) + c + b (i);
```

would unambiguously denote two calls to the two different routines.

Aliasing is exactly the opposite of overloading. Two names are aliases if theydenote the same entity at the same program point. This concept is especially relevant in the case of variables. Two alias variables share the same data object in the same referencing environment. Thus modification of the object under one name would make the effect visible, maybe unexpectedly, under the other.

Although examples of aliasing are quite common, one should be careful since this feature may lead to error prone and difficult to read programs. An exam-ple of aliasing is shown by the following C fragment:

```
inti;
int fun (int& a);
{
    . . .
    a = a + 1;
    printf ("%d", i);
    . . .
 }
main ( )
{
     . . .
     x = fun (i);
     . . .
 }
```

When function f is executed, names i and a in fun denote the same data object. Thus an assignment to a would cause the value of i printed by fun to differ from the value held at the point of call.

Aliasing can easily be achieved through pointers and array elements. For example, the following assignments in C

```
int x = 0;
int* i = &x;
int* j = &x;
```

would make *i, *j, and x aliases.

<u>**Run-time Structure:**</u>

Our discussion will show that languages can be classified in several categories, according to their execution-time structure.

Static languages: Exemplified by the early versions of FORTRAN and COBOL, these languages guarantee that the memory requirements for any program can be evaluated before program execution begins. Therefore, all the needed memory can be allocated before program execution.

Stack-based languages: Historically headed by ALGOL 60 and exemplified by the family of so-called Algol-like languages, this class is more demanding in terms of memory requirements, which cannot be computed at compile time. However, their memory usage is predictable and follows a last-in-first-out discipline: the lat-est allocated activation record is the next one to be deallocated.

Fully dynamic languages: These languages have un unpredictable memory usage; i.e, data are dynami-cally allocated only when they are needed during execution.

A hierarchy of lan-guages that are based on variants of the C programing language. They are named C1 through C3.

C1: A language with only simple statements

Let us consider a very simple programming language, called C1, which can be seen as a lexical variant of a subset of C, where we only have simple types and simple statements (there are no functions).

```
main ( )
{
    inti, j;
    get (i, j);
    while (i != j)
        if (i> j)
```

```
                i -= j;
            else
                i -= i;
        print (i);
    }
```

A C1 program is shown in above and its straightforward SIMPLESEM representation before the execution starts is shown in above. The D portion shows the activation record of the main program, which contains space for all variables that appear in the program.

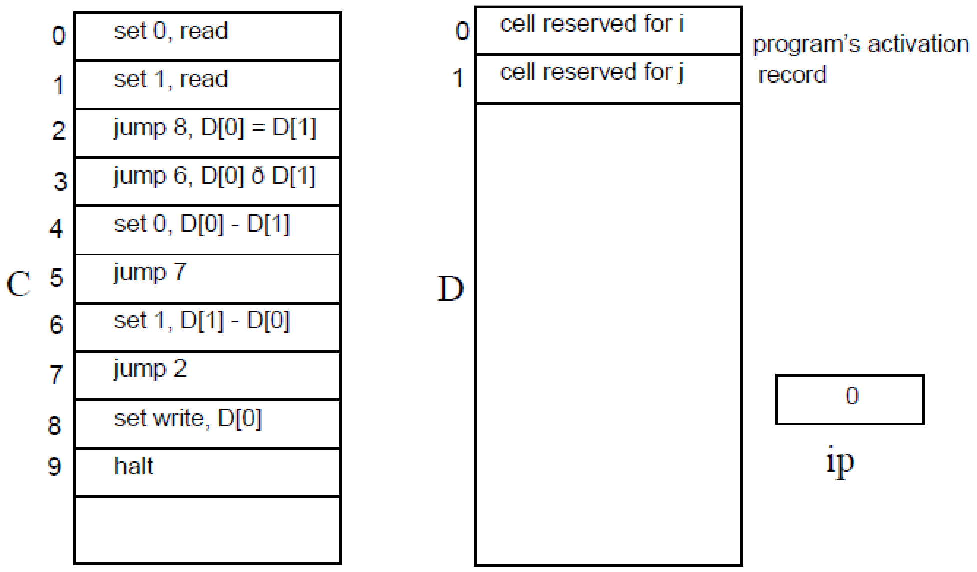


FIGURE 8. Initial state of the SIMPLESEM machine for the above C1 program

C2: Adding simple routines

Let us now add a new feature to C1. The resulting language, C2, allows routines to be defined in a program and allows routines to declare their own local data. A C2 program consists of a sequence of the following items:

- a (possibly empty) set of data declarations (*global data*);
- a (possibly empty) set of routine definitions and or declarations;
- a*main routine* (main ( )), which contains its local data declarations and a set of statements, that are automatically activated when the execution starts. The main routine cannot be called by other routines.

In bellow C2 program, whose main routine gets called initially, and causes routines beta and alpha to be called in a sequence.

```
inti = 1, j = 2, k = 3;
alpha ( )
{
inti = 4, l = 5;
...
i+=k+l;
...
};
beta ( )
{
int k = 6;
...
i=j+k;
alpha ( );
...
};
main ( )
{
...
beta ( );
...
}
```

Figure 9 shows the state of the SIMPLESEM machine after instruction i += kl of routine alpha has been executed. The first location of each activationrecord (offset 0) is reserved for the *return pointer*. Starting at location 1, space is reserved for the local variables.

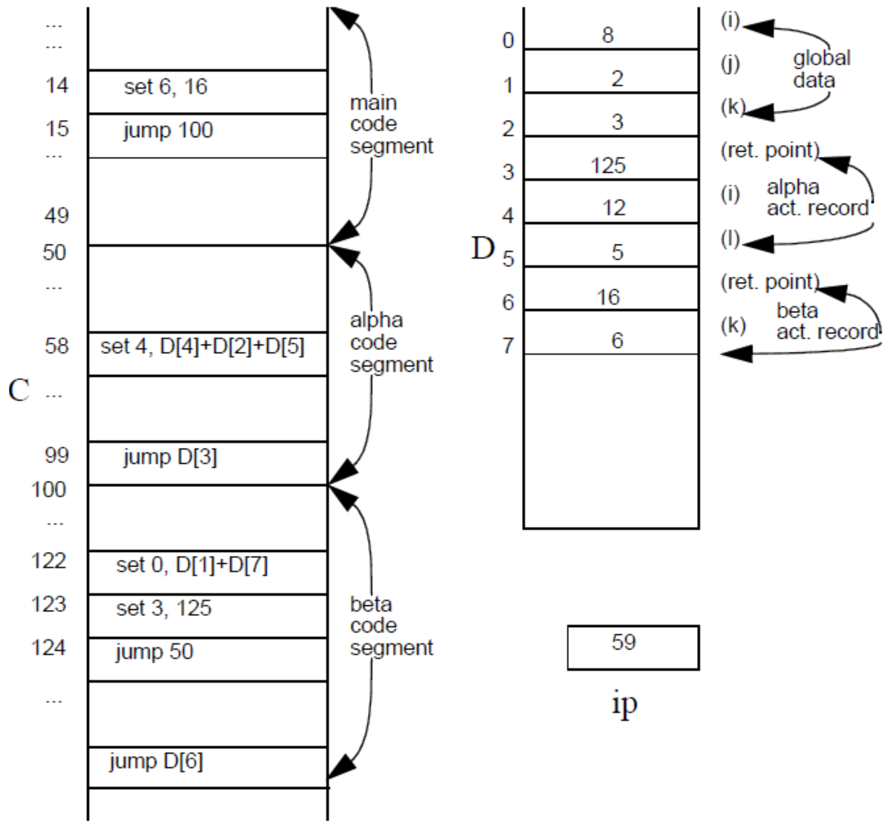


FIGURE 9. State of the SIMPLESEM executing the program of above

C3: Supporting recursive functions

Let us add two new features to C2: the ability of routines to call themselves (*direct recursion*) or to call one another in a recursive fashion (*indirect recur-sion*), and the ability of routines to return values, i.e., to behave as functions.These extensions define a new language, C3, which is illustrated in Figure 17 through an example.

```
int n;
int fact ( )
{
    intloc;
    if (n > 1) {
        loc = n--;
        returnloc * fact ( );
    }
    else
        return 1;
}
main ( )
{
    get (n);
    if (n >= 0)
        print (fact ( ));
    else
        print ("input error");
}
```

UNIT 2

Structuring the data

Built-in types and primitive types,

Programming languages organize data through the concept of *type*.Any programming language is equipped with a finite set of *built-in types* (or *predefined*) types, which normally reflect the behavior of the underlyinghardware.The built-in types of a programming language reflect the different views pro-vided by typical hardware. Examples of built-in types are:

- booleans, i.e., truth values TRUE and FALSE, along with the set of operations defined by Boolean algebra;
- characters, e.g., the set of ASCII characters;
- integers, e.g., the set of 16-bit values in the range <-32768, 37767>; and
- reals, e.g., floating point numbers with given size and precision.

In more detail, the following are advantages of built-in types:

Hiding of the underlying representation: This is an advantage provided by theabstractions of higher-level languages over lower-level (machine-level) languages.Invisibility of the underlying representation has the following benefits:

*Programming style.*The abstraction provided by the language increases programreadability by protecting the representation of objects from undisciplined manipulation.

Modifiability. The implementation of abstractions may be changed without affectingthe programs that make use of the abstractions.

Correct use of variables can be checked at translation time: If the type of each variable isknown to the compiler, illegal operations on a variable may be caught while the program is translated.

Resolution of overloaded operators can be done at translation time: For readabilitypurposes, operators are often overloaded. For example, + is used for both integer and real addition, * is used for both integer and real multiplication.

*Accuracy control.*In some cases, the programmer can explicitly associate a specificationof the accuracy of the representation with a type. For example, FORTRAN allows the user to choose between single and double-precision floating-point numbers. In C, integers can be short int, int, or long int.

Some types can be called *primitive* (or *elementary*). That is, they are not built from other types. Their values are atomic, and cannot be decomposed into simpler constituents. In most cases, built-in types coincide with primitive types, but there are exceptions. For example, in Ada both Character and String are predefined. Data of type String have constituents of type Character, how-ever. In fact, String is predefined as:

typeString **is array** (Positive **range** <>) **of** Character

It is also possible to declare new types that are elementary. An example is given by enumeration types in Pascal, C, or Ada. For example, in Pascal one may write:

typecolor = (white, yellow, red, green, blue, black);

The same would be written in Ada as

typecolor **is** (white, yellow, red, green, blue, black);

Similarly, in C one would write:

enum color {white, yellow, red, green, blue, black};

In the three cases, new constants are introduced for a new type. The constants are ordered; i.e., white < yellow < . . . < black. In Pascal and Ada, the built-in suc-cessor and predecessor functions can be applied to enumerations. For exam-ple, succ (yellow) in Pascal evaluates to red. Similarly.color’pred (red) in Adaevaluates to yellow.

<u>Data aggregates and typeconstructors,</u>

Programming languages allow the programmer to specify aggregations of elementary data objects and, recursively, aggregations of aggregates. They do so by providing a number of *constructors*. The resulting objects are called *compound objects*.

Older programming languages, such as FORTRAN and COBOL, provided only a limited number of constructors. For example, FORTRAN only pro-vided the array constructor; COBOL only provided the record constructor. In addition, through constructors, they simply provided a way to define a new *single* aggregate object, not a type. Later languages, such as Pascal, allowednew compound types to be defined by specifying them as aggregates of sim-pler types. In such a way, any number of instances of the newly defined aggregate can be defined. According to such languages, constructors can be used to define both aggregate objects and new aggregate types.

Cartesian product

The Cartesian product of n sets $A_1, A_2, \ldots, A_n$, denoted $A_1 \times A_2 \times \ldots \times A_n$, is a set whose elements are ordered n-tuples $(a_1, a_2, \ldots, a_n)$, where each a_k belongsto A_k. For example, regular polygons might be described by an integer– the number of edges– and a real– the length of each edge. A polygon would thus be an element in the Cartesian product integer x real.

Examples of Cartesian product constructors in programming languages are *structures* in C, C++, Algol 68 and PL/I, *records* in COBOL, Pascal, andAda. COBOL was the first language to introduce Cartesian products, which proved to be very useful in data processing applications.As an example of a Cartesian product constructor, consider the following C declaration, which defines a new type reg_polygon and two objects a_pol andb_pol;

```
structreg_polygon {
    intno_of_edges;
    floatedge_size;
};
structreg_polygonpol_a, pol_b = {3, 3.45};
```

The two regular polygons pol_a and pol_b are initialized as two equilateral tri-angles whose edge is 3.45. The notation {3, 3.45} is used to implicitly define a constant value (also called a *compound value*) of type reg_polygon (the polygon with 3 edges of length 3.45).

Finite mapping

A finite mapping is a function from a finite set of values of a domain type DT onto values of a range type RT. Such function may be defined in programminglanguages through the use of the mechanisms provided to define routines. This would encapsulate in the routine definition the law associating values of type RT to values of type DT. This definition is called *intensional*. In addition, programming languages, provide the *array* constructor to define finite map-pings as data aggregates. This definition is called *extensional*, since all the values of the function are explicitly enumerated. For example, the C declaration

```
char digits [10];
```

defines a mapping from integers in the subrange 0 to 9 to the set of characters, although it does not state which character corresponds to each element of the subrange. The following statements

```
for (i = 0; i< 10; ++i)
    digits [i] = ' ' ;
```

define one such correspondence, by initializing the array to all blank characters.This example also shows that an object in the range of the function is selected by *indexing*, that is, by providing the appropriate value in the domainas an index of the array. Thus the C notationcan be viewed as thedigits [i]application of the mapping to the argument i. Indexing with a value which is not in the domain yields an error.

C arrays provide only simple types of mappings, by restricting the domain type to be an integer subrange whose lower bound is zero. Other program-ming languages, such as Pascal, require the domain type to be an ordered dis-crete type. For example, in Pascal, it is possible to declare

```
var x: array [2. .5] of integer;
```

which defines x to be an array whose domain type is the subrange 2. .5.

Similarly, in Ada one might write

```
X: array (INTEGER range 2. .6) of INTEGER :=  (0, 2, 0, 5, -33);
```

to define an array whose index is in the subrange 2. .5, where X(2) = 0, X(3) = 2,X(4) = 0, X(5) = 5, X(6) = -33.
It is interesting to note that Ada uses brackets "(" and ")" instead of "[" and "]" to index arrays.

Notice that an array element can– in turn– be an array. This allows multidi-mensional arrays to be defined. For example, the C declaration

```
int x[10][20];
```

declares an integer rectangular array of 10 rows and 20 columns.

Union and discriminated union

Cartesian products defined in Section 3.2.1 allow an aggregate to be constructed through the *conjunction* of its fields. For example, we saw the exam-ple of a polygon, which was represented as an integer (the number of edges) *and* a real (the edge size). In this section we explore a constructor whichallows an element (or a type) to be specified by a *disjunction* of fields.

For example, suppose we wish to define the type of a memory address for a machine providing both absolute and relative addressing. If an address is relative, it must be added to the value of some INDEX register in order to access the corresponding memory cell. Using C, we can declare

```
union address {
    shortint offset;
    long unsigned int absolute;
};
```

The declaration is very similar to the case of a Cartesian product. The difference is that here fields are mutually exclusive.

Values of type address must be treated differently if they denote offsets or absolute addresses. Given a variable of type address, however, there is no automatic way of knowing what kind of value is currently associated with the variable (i.e., whether it is an absolute or a relative address). The burden of remembering which of the fields of the union is current rests on the program-mer. A possible solution is to consider an address to be an element of the fol-lowing type:

```
structsafe_address {
    address location;
    descriptor kind;
};
```

wheredescriptor is defined as an enumeration

```
enum descriptor {abs, rel};
```

A safe address is defined as composed of two fields: one holds an address, the other holds a descriptor. The descriptor field is used to keep track of the cur-rent address kind. Such a field must be updated for each assignment to the corresponding location field.

Powerset

It is often useful to define variables whose value can be any subset of a set of elements of a given type T. The type of such variables is powerset (T), the set of all subsets of elements of type T. Type T is called the *base type*. For example, suppose that a language processor accepts the following set O of options

- LIST_S, to produce a listing of the source program;
- LIST_O, to produce a listing of the object program;
- OPTIMIZE, to optimize the object code;
- SAVE_S, to save the source program in a file;
- SAVE_O, to save the object program in a file;
- EXEC, to execute the object code.

Sequencing

A sequence consists of any number of occurrences of elements of a certain component type CT. The important property of the sequencing constructor is that the number of occurrences of the component is unspecified; it therefore allows objects of arbitrary size to be represented.the best example is the file constructor of Pascal, which models the conventional data processing concept of a sequential file. Elements of the file can be accessed sequentially, one after the other. Modifications can be accomplished by appending a new values at the end of an existing file. Files are provided in Ada through standard libraries, which support both sequential and direct files.

Arrays and recursive list definitions (defined next) may be used to represent sequences, if they can be stored in main memory. If the size of the sequence does not change dynamically, arrays provide the best solution. If the size needs to change while the program is executing, flexible arrays or lists must be used. The C++ standard library provides a number of sequence implementations, including vector and list.

User-defined types and abstract data types,

Modern programming languages provide many ways of defining new types, starting from built-in types.

For example, after the C declaration which introduces a new type name complex

```
struct complex {
    floatreal_part, imaginary_part;
}
```

any number of instance variables may be defined to hold complex values:

```
complex a, b, c, . . .;
```

By providing appropriate type names, program readability can be improved. In addition, by factoring the definition of similar data structures in a type declaration, modifiability is also improved. A change that needs to be applied to the data structures is applied to the type, not to all variable declarations. Factorization also reduces the chance of clerical errors and improves consistency.

The ability to define a type name for a user defined data structure is only a first step in the direction of supporting data abstractions.we need a construct to define abstract data types. An *abstract data type* is a new type for which we can define the opera-tions to be used for manipulating instances, while the data structure that implements the type is hidden to the users. In what follows we briefly review the constructs provided by C++

Abstract data types in C++

Abstract data types can be defined in C++ through the class construct. A class encloses the definition of a new type and explicitly provides the operations that can be invoked for correct use of instances of the type. As an example, Figure 33 shows a class defining the type of the geometrical concept of point.

```
class point {
    int x, y;
public:
    point (int a, int b) { x = a; y = b; }
    voidx_move (int a) { x += a; }
    voidy_move (int b ){ y += b; }
    void reset ( ) { x = 0; y = 0; }};
```

<u>Type Systems:</u>

Types are a fundamental semantic concept of programming languages. More-over, programming languages differ in the way types are defined and behave, and typing issues are often quite subtle.

Type is defined as a set of values and a set of operations that can be applied to such values. As usual, since values in our context are stored somewhere in the memory of a computer, we use the term *object* (or *data object*) to denote both the storage and the stored value. The operations defined for a type are the only way of manipulating its instance objects: they protect data objects from any illegal uses. Any attempt to manipulate objects with illegal operations is a *type error*. A program is said to be *type safe* (or *type secure*) if all operationsin the program are guaranteed to always apply to data of the correct type, i.e., no type errors will ever occur.

Static versus dynamic program checking

Errors can be classified in two categories: language errors and application errors. *Language errors* are syntactic and semantic errors in the use of the programming language. *Application errors* are deviations of the program behavior with respect to specifications (assuming specifications capture the required behavior correctly). The programming language should facilitate both kinds of errors to be identified and removed.

Error checking can be accomplished in different ways, that can be classified in two broad categories: static and dynamic. *Dynamic checking* requires the program to be executed on sample input data. *Static checking* does not.

Strong typing and type checking

The type system of a language was defined as the set of rules to be followed to define and manipulate program data. Such rules constrain the set of legal programs that can be written in a language. The goal of a type system is to prevent the writing of type unsafe programs as much as possible. A type sys-tem is said to be *strong* if it guarantees type safety; i.e., programs written by following the restrictions of the type system are guaranteed not to generate type errors. A language with a strong type system is said to be a *stronglytyped language*. If a language is strongly typed, the absence of type errorsfrom programs can be guaranteed by the compiler. A type system is said to be *weak* if it is not strong. Similarly, a *weakly typed language* is a language thatis not strongly typed.

Type conversions

Suppose that an object of type T1 is expected by some operation at some point of a program. Also, suppose that an object of type T2 is available and we wish to apply the operation to such object. If T1 and T2 are compatible according to the type system, the application of the operation would be type correct. If they are not, one might wish to apply a type conversion from T2 to T1 in order to make the operation possible.

C provides a simple coercion system. In addition, explicit conversions can be applied in C using the *cast* construct. For example, a cast can be used to over-ride an undesirable coercion that would otherwise be applied in a given con-text. For example, in the above assignment, one can force a conversion of z to intby writing

```
x = x + (int) z;
```

Such an explicit conversion in C is semantically defined by assuming that the expression to be converted is implicitly assigned to an unnamed variable of the type specified in the cast, using the coercion rules of the language.

Types and subtypes

If a type is defined as a set of values with an associated set of operations, a subtype can be defined to be a subset of those values (and, for simplicity, the same operations).

If ST is a subtype of T, T is also called ST's supertype (or parent type). We assume that the operations defined for T are automatically inherited by ST. A language supporting subtypes must define:

1. a way to define subsets of a given type;
2. compatibility rules between a subtype and its supertype.

Pascal was the first programming language to introduce the concept of a sub-type, as a subrange of any discrete ordinal type (i.e., integers, boolean, character, enumerations, or a subrange thereof). For example, in Pascal one may define natural numbers and digits as follows:

```
typenatural = 0. .maxint;
     digit = 0. .9;
     small = -9. .9;
```

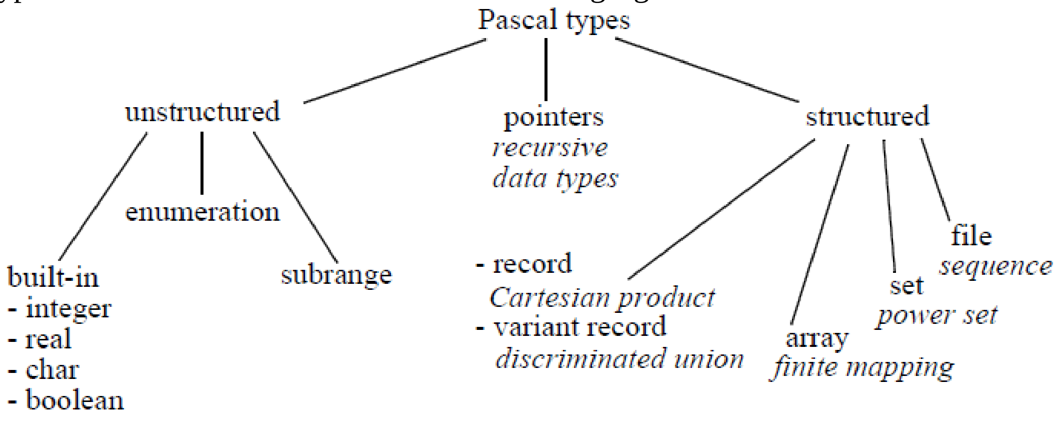
wheremaxint is the maximum integer value representable by an implementa-tion.

<u>The typeStructure of representative languages,</u>

This description provides an overall hierarchical taxonomy of the features provided by each language for data structuring.

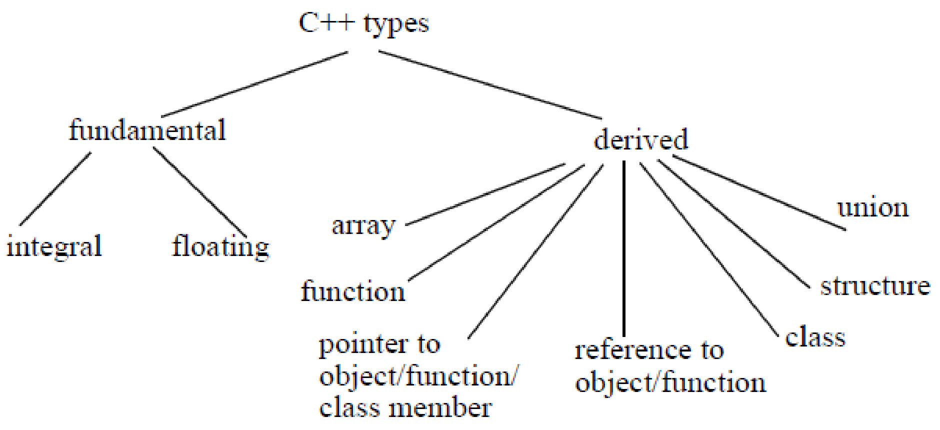
Pascal

The type structure of Pascal is describedin following figure



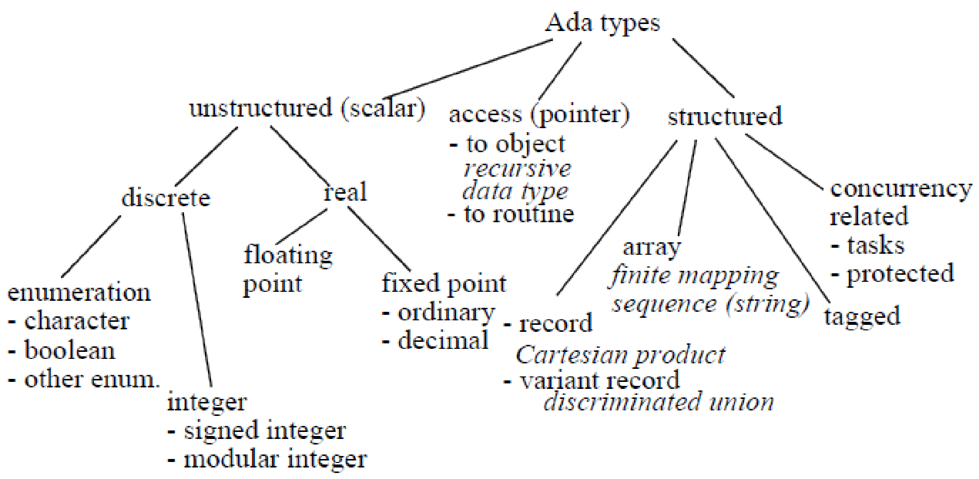
C++

The type structure of C++ is givenin following figure



Ada

The type structure of Ada is describedin following figure

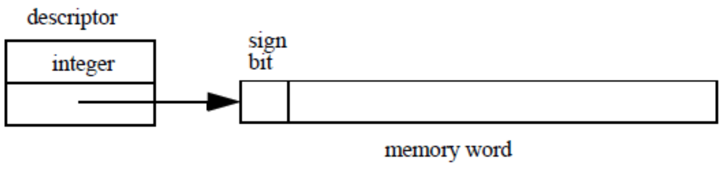


<u>Implementation Models</u>

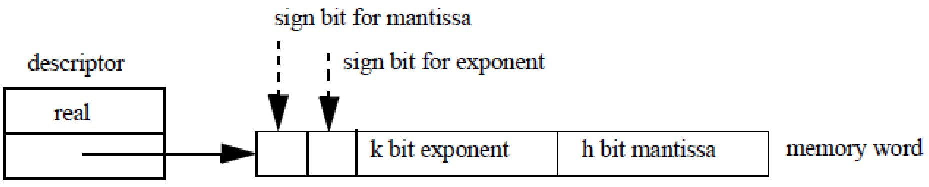
Data will be represented by a pair consisting of a *descriptor* and a *data object*. Descriptors contain the most relevant attributesthat are needed during the translation process. Additional attributes might be appropriate for specific purposes.

Built-in and enumerations

Integers and reals are hardware-supported on most conventional computers,In a language like C, these are reflected by long and short prefixes. Integer and real variables may be represented as shown in following Figure.



Representation of integer variables



Representation of floating point variables

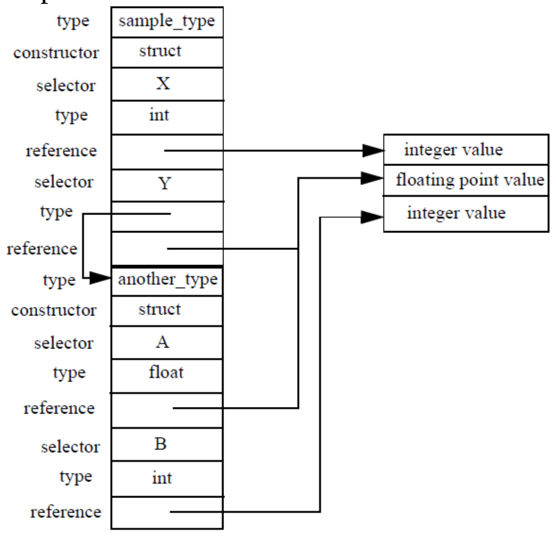
Structured types

Cartesian product

The standard representation of a data object of a Cartesian product type is a sequential layout of components. The descriptor contains the Cartesian prod-uct type name and a set of triples (name of the selector, type of the field, reference to the data object) for each field. If the type of the field is not a built-in type, the type field points to an appropriate descriptor for the field.

```
typeanother_type Y is struct{
    float A;
    int B;
};
typesample_type is struct {
    int X;
    another_type Y;
};
sample_type X;
```

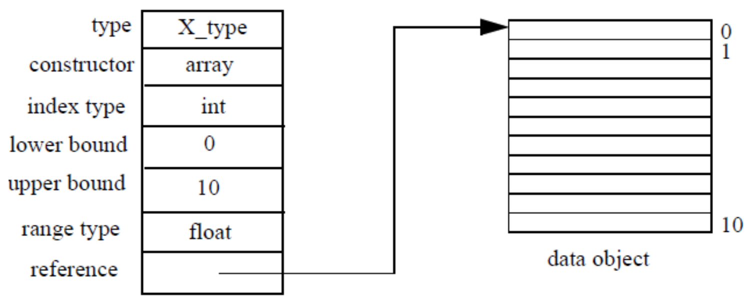
Representation of Cartesian product



Finite mapping

A conventional representation of a finite mapping allocates a certain number of storage units (e.g., words) for each component. The descriptor contains the finite-mapping type name; the name of the domain type, with the values of the lower and upper bound; the name of the range type (or the reference to its descriptor); the reference to the first location of the data area where the data object is stored.For example, given the declarations

```
typeX_type is float array [0. .10];
X_type X;
```

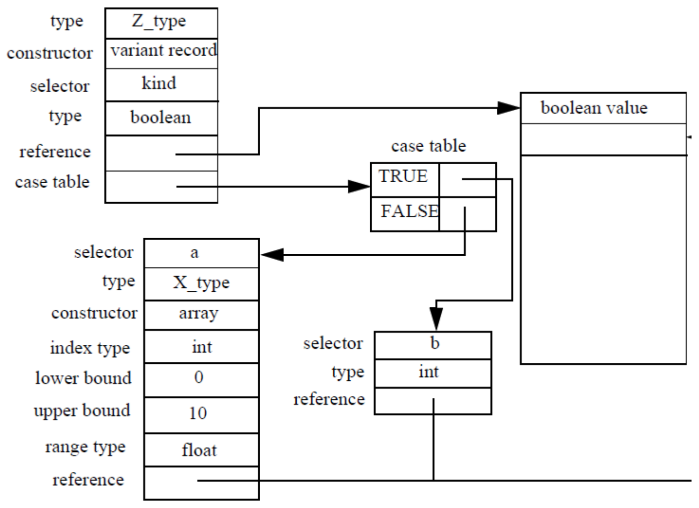


Representation of Finite mapping

Union and discriminated union

Union types do not require any special new treatment to be represented. A variable of a union type has a descriptor that is a sequence of the descriptorsof the component types. nstances of the values of the component types share the same storage area.
As an example, the reader may consider the following Pascal-like fragment.

```
        typeX_type is float array [0. .10];
        typeZ_type = record
                              case kind:  BOOLEAN of
                                 TRUE: (a: integer);
                                 FALSE: (b: X_type)
                  end
```



Representation of *Union and discriminated union*

UNIT 3

Structuring the Computation

Programs are often decomposed into units. For example, routines provide a way of hierarchically decomposing a program into units representing new complex operations. Once program units are constructed, it becomes necessary to structure the flow of computation among such units.

Expressions and statements

Expressions define how a value can be obtained by combining other values through operators. The values from which expressions are evaluated are either denoted by a literal, as in the case of the real value 57.73, or they are the r_value of a variable.

Operators appearing in an expression denote mathematical functions. They are characterized by their *aritiy* (i.e., number of operands) and are invoked using the function's signature. A unary operator is applied to only one operand. A binary operator is applied to two operands. In general, a n-ary operator is applied to n operands. For example, ' -' can be used as a unary operator to transform say the value of a positive expression into a negative value. In general, however, it is used as a binary operator to subtract the value of one expression from the value of another expression. Functional routine invocations can be viewed as n-ary operators, where n is the number of parameters.

Regarding the operator's notation, one can distinguish between infix, prefix, and postfix. *Infix notation* is the most common notation for binary operators: the operator is written between its two operands, as in x + y. Postfix and prefix notations are common especially for non-binary operators. In *prefix notation*, the operator appears first, and then the operands follow. This is the conventional form of function invocation, where the function name denotes the operator. In *postfix notation* the operands are followed by the corresponding operator. Assuming that the arity of each operator is fixed and known, expres-sions in prefix and postfix forms may be written without resorting to parentheses to specify subexpressions that are to be evaluated first. For example, the infix expression a * (b + c) can be written in prefix form as * a + b c and in postfix form as a b c + *

In C, the increment and decrement unary operators ++ and -- can be written both in prefix and in postfix notation. The semantics of the two forms, how-ever, is different; that is, they denote two distinct operators. Both expressions ++k and k++ have the side effect that the stored value of k is incremented by one. In the former case, the value of the expression is the value of k incre-mented by one (i.e., first, the stored value of k is incremented, and then the value of k is provided as the value of the expression). In the latter case, the value of the expression is the value of k before being incremented.

Although the programmer may use parentheses to explicitly group subexpressions that must be evaluated first, programming languages complicate matters by introducing their own conventions for operator associatively and precedence. For example, the convention adopted by most languages is such that a + b * c is interpreted implicitly as a + (b * c) i.e., multiplication has precedence over binary addition (as in standard mathematics). However, consider the Pascal expression a = b < c and the C expression a == b < c

In Pascal, operators < and = have the same precedence, and the language specifies that application of operators with the same precedence proceeds left to right. The meaning of the above expression is that the result of the equality test (a=b), which is a boolean value, is compared with the value of c (which must be a boolean variable). In Pascal, FALSE is assumed to be less than TRUE, so the expression yields TRUE only if a is not equal to b, and c is TRUE; it yelds FALSE in all other cases. For example, if a, b and c are all FALSE, the expression yields FALSE.

In C, operator "less than" (<) has higher precedence than "equal" (==). Thus, first b < c is evaluated. Such partial result is then compared for equality with the value of a. For example, assuming a = b = c = false (represented in C as zero), the evaluation of the expression yields 1, which in C stands for true.

Some programming languages support the ability of writing conditional expressions, i.e., expressions that are composed of subexpressions, of which only one is to be evaluated, depending on the value of a condition. For exam-ple, in C one can write (a > b) ? a : b which would be written in a perhaps more conventionally understandable form in ML as if a > b then a else b to yield the maximum of the values of a and b.

ML allows for more general conditional expressions to be written using the "case" constructor, as shown by the following simple example.

```
case x of
   1 => f1 (y)
 | 2 => f2 (y)
 | _ => g (y)
```

In the example, the value yielded by the expression is f1 (y) if x = 1, f2 (y) if x = 2, g (y) otherwise.

Conditional execution and iteration

Conditional execution of different statements can be specified in most languages by the if statement. Let us start with the example of the if statement as originally provided by Algol 60. Two forms are possible, as shown by the following examples:

```
if i = 0                    if i = 0
   then i := j;                then i := j
                               else begin
```

```
                        j := j - 1

               i := i + 1;
                 end
```

In the first case, no alternative is specified for the case i ¦0, and thus nothing happens if i ¦0. In the latter, two alternatives are present. Since the case where i ¦0 is described by a sequence, it must be made into a compound statement by bracketing it between begin and end.

Choosing among more than two alternatives using only if-then-else state-ments may lead to awkward constructions, such as

```
if a
  then S1
  else
     if b
       then S2
       else
```

```
            if c
                then S3
                else S4
            end
        end
    end
```

To solve this syntactic inconvenience, Modula-2 has an else-if construct that also serves as an end bracket for the previous if. Thus the above fragment may be written as

```
if a
    then S1
else if b
    then S2
else if c
    then S3
else S4
end
```

C, Algol 68, and Ada provide similar abbreviations.

Most languages also provide an ad-hoc construct to express multiple-choice selection. For example, C++ provides the switch construct, illustrated by the following fragment:

```
switch (operator) {
    case ’+’:
        result = operand1 + operand2;
        break;
    case ’*’:
        result = operand1 * operand2;
        break;
    case ’- ’:
        result = operand1 - operand2;
        break;
    case ’/’:
        result = operand1 / operand2;
        break;
    default:
        break; --do nothing
};
```

The same example may be written in Ada as

```
case OPERATOR is
    when ’+’ => result = operand1 + operand2;
    when ’*’ => result = operand1 * operand2;
    when ’-’ => result = operand1 - operand2;
    when ’/’ => result = operand1 / operand2;
    when others => null;
end case
```

In Ada, after the selected branch is executed, the entire case statement termi-nates.

Iteration allows a number of actions to be executed repeatedly. Most programming languages provide different kinds of *loop constructs* to define iter-ation of actions (called the *loop body*). Often, they distinguish between loops where the number of repetitions is known at the start of the loop, and loops where the body is executed repeatedly as long as a condition is met. The former kind of loop is usually called a *for loop*; the latter is often called the *while loop*.

For-loops are named after a common statement provided by languages of the Algol family. For statements define a *control variable* which assumes all values of a given predefined sequence, one after the other. For each value, the loop body is executed.

Pascal allows iterations where control variables can be of any ordinal type: integer, boolean, character, enumeration, or subranges of them. A loop has the following general appearance:

```
for loop_ctr_var := lower_bound to upper_bound do statement
```

A control variable assumes all of its values from the lower to the upper bound. The language prescribes that the control variable and its lower and upper bounds must not be altered in the loop. The value of the control vari-able is also assumed to be undefined outside the loop.

As an example, consider the following fragment:

```
type day = (sun, mon, tue, wed, thu, fri, sat);
var week_day: day;

for week_day := mon to fri do . . .
```

As another example, let us consider how for-loops can be written in C++, by examining the following fragment, where the loop body is executed for all values of i from 0 to 9

```
for (int i = 0; i < 10; i++) {. . .}
```

The statement is clearly composed of three parts: an initialization and two expressions. The initialization provides the initial state for the loop execution. The first of the two expressions specifies a test, made before each iteration, which causes the loop to be exited if the expression becomes zero (i.e., false). The second specifies the incrementing that is performed after each iteration. In the example, the statement also declares a variable i. Such variable's scope extends to the end of the block enclosing the for statement.

While loops are also named after a common statement provided by languages of the Algol family. A while loop describes any number of iterations of the loop body, including zero. They have the following general form

```
while condition do statement
```

For example, the following Pascal fragment describes the evaluation of the greatest common divisor of two variables a and b using Euclid's algorithm

```
while a ¦b do
```

```
begin
   if a > b then
       a := a - b
    else
        b := b - a
end
```

The end condition (a ¦ b) is evaluated before executing the body of the loop. The loop is exited if a is equal to b, since in such case a is the greatest common divisor. Therefore, the program works also when it is executed in the special case where the initial values of the two variables are equal.

In C++, while statements are similar. The general form is:

```
while (expression) statement
```

Often languages provide another similar kind of loop, where the loop control variable is checked at the end of the body. In Pascal, the construct has the fol-lowing general form

```
repeat
    statement
until condition
```

In a Pascal repeat loop, the body is iterated as long as the condition evaluates to false. When it becomes true, the loop is exited.

C++ provides a do-while statement which behaves in a similar way:

```
do statement while (expression);
```

In this case the statement is executed repeatedly until the value of the expres-sion becomes zero (i.e., the condition is false).

Ada has only one general loop structure, with the following form

```
iteration_specification loop
    loop_body
end loop
```

C++ provides a break statement, which causes termination of the smallest enclosing loop and passes control to the statement following the terminated statement, if any. It also provides a continue statement, which causes the termi-nation of the current iteration of a loop and continuation from the next itera-tion (if there is one). A continue statement can appear in any kind of loop (for loop, and both kinds of while loops).

Routines

Routines are a program decomposition mechanism which allows programs to be broken into several units. Routine calls are control structures that govern the flow of control among program units.

Most languages distinguish between two kinds of routines: procedures and functions. A *procedure* does not return a value: it is an abstract command which is called to cause some desired state change. A *function* corresponds to its mathematical counterpart: its activation is supposed to return a value, which depends on the value of the transmitted parameters.

Pascal provides both procedures and functions. It allows formal parameters to be either by value or by reference. It also allows procedures and functions to be parameters, as shown by the following example of a procedure header:

```
procedure example (var x: T; y: Q; function f (z: R): integer);
```

In the example, x is a by-reference parameter of type T; y is a by-value param-eter of type Q; f is a function parameter which takes one by-value parameter z of type R and returns an integer.

Ada provides both procedures and functions. Parameter passing mode is spec-ified in the header of an Ada routine as either in, out, or in out. If the mode is not specified, in is assumed by default. A formal in parameter is a constant which only permits reading of the value of the corresponding actual parameter. A formal out parameter is a variable and permits updating of the value of the associated actual parameter. In the implementation, parameters are passed either by copy or by reference. Ada defines the program to be erroneous; but, unfortunately, the error can only be discovered at run time.

In C all routines are functional, i.e., they return a value, unless the return type is void, which states explicitly that no value is returned. Parameters can only be passed by value. It is possible, however, to achive the effect of call by ref-erence through the use of pointers. For example, the following routine

```
void proc (int* x, int y);
{
    *x = *x + y;
}
```

increments the object referenced by x by the value of y. If we call proc as follows

```
proc (&a, b);  /*  &a means the address of a */
```

x is initialized to point to a, and the routine increments a by the value of b.

C++ introduced a way of directly specifying call by reference. This frees the programmer from the lower level use of pointers to simulate call by reference.

The previous example would be written in C++ as follows.

```
void proc (int& x, int y);
{
    x = x + y;
}

proc (a, b); -- no address operator is needed in the call
```

While Pascal only allows routines to be passed as parameters, C++ and Ada get closer to treating routines as first-class objects. For example, they provide pointers to routines, and allow pointers to be bound dynamically to different routines at run time.

Exceptions

Programmers often write programs under the optimistic assumption that nothing will go wrong when the program executes. Unfortunately, however, there are many reasons which may invalidate this assumption. For example, it may happen that under certain conditions an array is indexed with a value which exceeds the declared bounds. An arithmetic expression may cause a division by zero, or the square root operation may be executed with a negative argument. A request for new memory allocation issued by the run-time system might exceed the amount of storage available for the program execution. Or, finally, an embedded application might receive a message from the field which overrides a previously received message, before this message has been handled by the program.

To cope with this problem, programming languages provide features for exception handling. According to the standard terminology, an *exception* denotes an undesirable, anomalous behavior which supposedly occurs rarely. The language can provide facilities to define exceptions, recognize them, and specify the response code that must be executed when the exception is raised (*exception handler*).

Earlier programming languages (except PL/I) offered no special help in properly handling exceptional conditions. Most modern languages, however, provide systematic exception-handling features. With these features, the concern for anomalies may be moved out of the main line of program flow, so as not to obscure the basic algorithm.

To define exception handling, the following main decisions must be taken by a programming language designer:

- What are the exceptions that can be handled? How can they be defined?
- What units can raise an exception and how?
- How and where can a handler be defined?
- How does control flow after an exception is raised in order to reach its handler?
- Where does control flow after an exception has been handled?

The solutions provided to such questions, which can differ from language to language, affect the semantics of exception handling, its usability, and its ease of implementation. In this section, we will analyze the solutions provided by C++, Ada, Eiffel, and ML. The exception handling facilities of PL/I and CLU are shown in sidebars.

Exception handling in Ada

. .

Ada provides a set of four predefined exceptions that can be automatically trapped and raised by the underlying run-time machine:

- Constraint_Error: failure of a run-time check on a constraint, such as array index out of bounds, zero right operand of a division, etc.;
- Program_Error: failure of a run-time check on a language rule. For example, a function is required to complete normally by executing a return statement which transmits a result back to the caller. If this does not happen, the exception is raised;
- Storage_Error: failure of a run-time check on memory avaliability; for example, it may be raised by invocation of new;
- Tasking_Error: failure of a run-time check on the task system

A program unit can declare new exceptions, such as

```
Help: exception;
```

which can be explicitly raised in their scope as

```
raise Help;
```

Once they are raised, built-in and programmer-defined exceptions behave in exactly the same way. Exception handlers can be attached to a subprogram body, a package body, or a block, after the keyword exception. For example

```
begin --this is a block with exception handlers
    ... statements ...
    exception    when Help =>handler for exception Help
                 when Constraint_Error => handler for exception
                         Constraint_Error, which might be raised by a
                         division by zero

                 when others => handler for any other exception that is
                         not Help nor Constraint_Error
end;
```

In the example, a list of handlers is attached to the block. The list is prefixed by the keyword **exception**, and each handler is prefixed by the keyword **when**.

Exception handling in C++

Exceptions may be generated by the run-time environment (e.g., due to a division by zero) or may be explicitly raised by the program. An exception is raised by a throw instruction, which transfers an object to the corresponding handler. A handler may be attached to any piece of code (a block) which needs to be fault tolerant. To do so, the block must be prefixed by the key-word try. As an example, consider the following simple case:

```
class Help {. . . }; // objects of this class have a public attribute "kind" of type enumeration

 // which describes the kind of help requested, and other public fields which

// carry specific information about the point in the program where help

//is requested
class Zerodivide { }; // assume that objects of this class are generated by the run-time sys-

tem

. . .

try {

//fault tolerant block of instructions which may raise help or zerodivide exceptions
. . .

}

catch (Help msg) {

//                    handles a Help request brought by object
msg switch (msg.kind) {

case MSG1:

    . . .; case MSG2:

. . .;

. . .

}

. . .

}

catch (Zerodivide) {
```

```
//handles a zerodivide situation
  . . .

  }
```

Suppose that the above try block contains the statement throw Help (MSG1); A throw expression causes the execution of the block to be abandoned, and control to be transferred to the appropriate handler. It also initializes a temporary object of the type of the operand of throw and uses the temporary to ini-tialize the variable named in the handler. In the example, actually invokes the constructor of class Help passing a parameter which is used by the constructor to initialize field kind. The temporary object so created is used to initialize the formal parameter msg of the matching catch, and control is then transferred to the first branch (case MSG1) of the switch in the first handler attached to the block.

C++ routines may list in their interface the exeception they may raise. This feature allows a programmer to state the intent of a routine in a precise way, by specifying both the expected normal behavior (the data it can accept and return), and its abnormal behaviors. For example

```
void foo ( ) throw (Help, Zerodivide);
```

might be the interface of a function foo which is called within the above fault tolerant block. Knowing that the used function foo may indeed raise excep-tions, the client code may guard against anomalous behaviors by providing appropriate exception handling facilities,

```
enum Help {MSG1, MSG2, ...};
```

and the corresponding catch statement would be rewritten as

```
catch (Help msg) {
    switch (msg) {
    case MSG1:
        . . .;
    case MSG2:
        . . .;
    . . .
    }
    . . .
}
```

Other interesting ways of organizing exceptions can be achieved by organiz-ing the corresponding classes according to subtype hierarchies, by means of subclasses

Exception handling in Eiffel

The features provided by Eiffel to support exception handling have been strongly influenced by a set of underlying software design principles that pro-grammers should follow. A key notion of such design principles is called the *contract*.. Semantically, they are specified by the preconditions and postconditions of every exported routines and by the invariant condition. Once the program compiles correctly, syntactic obligations are guaranteed to have been verified. What may happen, however, is that semantic obligations are not fulfilled during execution. This is how exceptions may arise in Eiffel.

Thus, exceptions may arise in Eiffel because an assertion is violated (assum-ing that the program has been compiled under the option that sets runtime checking on). They can also arise because of anomalous states caught by the underlying abstract machine (memory exhausted, dereferencing an uninitial-ized pointer, ...). Finally, they can arise because a called routine fails (see below for what this means).

This strategy can be stated in Eiffel according to the following scheme:

```
try_several_methods is
local
   i: INTEGER;
   --it is automatically initialized to 0
do
   try_method
(i); rescue
   i := i + 1;
   if i < max_trials then --
      max_trials is a
      constant retry
   end
```

Exception Handling in PL/I

PL/I was the first language to introduce exception handling. Exceptions are called CONDITIONS in PL/I. Exception handlers are declared by ON state-ments:

```
ON CONDITION (exception_name) exception_handler
```

where exception_handler can be a simple statement or a block. An exception is explicitly raised by the statement

```
SIGNAL CONDITION (exception_name);
```

The language also defines a number of built-in exceptions and provides sys-tem-defined handlers for them. Built-in exceptions are automatically raised by the execution of some statements (e.g., ZERODIVIDE is raised when a divide by zero is attempted). The action performed by a system-provided han-dler is specified by the language. This action can be redefined, however, as with user-defined exceptions:

```
ON ZERODIVIDE BEGIN;
   . . .
END;
```

PL/I does not allow the programmer to pass any information from the point raising the exception to the exception handler. If this is necessary, the pro-grammer must resort to global variables, which can be an unsafe program-ming practice.

Exception handling in ML

The functional language ML allows exceptions to be defined, raised, and han-dled. There are also exceptions that are predefined by the language and raised automatically by the runtime machine while the program is being executed.

As an example, the following declaration introduces an exception

```
exception Neg
```

which can be raised subsequently in the following function declaration

```
fun fact (n) =
    if n < 0 then raise Neg
    else if n = 0 then 1
    else n * fact (n - 1)
```

A call such as fact (-2) would cause the evaluation of the function to be aban-doned, the exception raised and, since no handler is provided, the program to stop by writing the message "Failure: Neg".

Suppose we wish to handle the exception by returning 0 when the function is called with a negative argument. This can be done, for example, by defining the following new function

```
fun fact_0 (n) = fact (n) handle Neg => 0;
```

Pattern matching

Pattern matching is a high level way of stating conditions, based on which, different actions are specified to occur. Pattern matching is the most important control structure of the string manipulation programming language SNOBOL4. Pattern matching is also provided by most modern functional programming languages, like ML, Miranda, SASL, and is also pro-vided by the logical language PROLOG and by rule-based systems.

Let us start by discussing the following simple definitions of a data type and a function:

```
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
fun day_off  (Sun) = true
    |day_off (Sat )= true
    |day_off  ( _  ) = false
```

In the example, function day_off is defined by a number of cases. Depending on the value of the parameter with which the function will be invoked, the appropriate case will be selected. Cases are checked sequentially, from the first one on. If the first two cases are not matched by the value of the parame-ter with which the function is called, the third alternative will be selected, since the so-called wild card "_" matches any argument.

As another example, consider the following function definition:

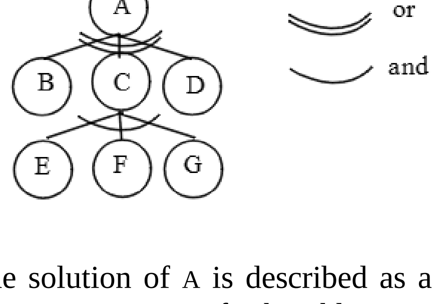
```
fun reverse (nil) = nil
    | reverse (head::tail) = reverse(tail) @ [head]
```

In this case, if the argument is an empty list, then reverse is defined to return the empty list. Otherwise, suppose that the argument is the list [1, 0, 5, 99, 2]. As a result of pattern matching [1, 0, 5, 99, 2] with head::tail, head is bound to 1 and tail is bound to [0, 5, 99, 2]. Thus the result of reverse is the concatenation (operator @) of reverse ([0, 5, 99, 2]) with the list [1].

Nondeterminism and backtracking

Problem solutions can often be described via and-or decompositions into sub-problems. For example, to solve problem A, one needs to solve either B, C, or D; to solve– say– C, one needs to solve E, F, and G. This can be represented as an *and/or tree* (see Figure 52). Node A, which has no incoming arcs, is called a *root node*; nodes B, D, E, F, and G, which have no exiting arcs, are called *leaf nodes*. And/or decompositions can also be described– in the hypothetical syn-tax of some programming language– as

```
A if B or
   C or
   D;
C if E and
F and
G;
```



The solution of A is described as a *disjunction* of subproblems; the solution of C is described as a *conjunction* of subproblems. We can further assume B, D, E, F, and G to be problem solving routines, which can terminate in either a success or a failure state.

If the order in which subproblems are solved is unspecified (and irrelevant as far as the problem statement is concerned), we say that the program is *nonde-terministic*. In the example, this means that the order in which B, C, or D are tried does not matter. Similarly, the way in which E, F, and G are tried does not matter. The only thing that matters is that a solution be found, if it exists, or a notification of failure is delivered to the request to solve A, if no solution exists. The latter case happens if all three subproblems in the disjunct fail, which means also that at least one of the subproblems of the conjunction failed.

One solution strategy is to explore the and/or tree in *parallel*, in order to make the search for the solution time efficient. In such a case, subproblems B, D, E, F, and G would be solved in parallel. Another more classical strategy for a sequential implementation of the search is based on *backtracking*. Backtracking means that to find a solution, a choice is made at each branch of the tree. The choice can be fixed, arbitrary, or based on some heuristic knowledge of the problem being solved. If a subproblem solution fails, backtracking implies that another possible subproblem solution be tried. This ensures that the overall problem solution fails only if there is no way of solving the prob-lem without failure. Thus, through backtracking, one needs to guarantee com-pleteness of the search.

Event-driven computations

In some cases, programs are structured conveniently as *reactive* systems, i.e., systems where certain *events* occurring in the environment cause certain pro-gram fragments to be executed. An example is provided by modern user inter-faces where a number of small graphical devices (called *widgets*) are often displayed to mediate human-computer interaction. By operating on such widgets (e.g., by clicking the mouse on a push-button) the user generates an event. The event, in turn causes a certain application fragment to be executed. Execution of such a fragment may cause further screen layouts to be gener-ated with a new context of available widgets on it. The events that can be gen-erated in any given state are defined by the context.

The entire application can be viewed as a system which reacts to events by dispatching them to the appropriate piece of code that is responsible for handling the event. As a consequence, the application is structured as a set of fragments that are responsible for handling specific events.

This conceptual view of an application can be viewed as a way of structuring its high-level design, which would then need to be detailed by a conventional implementation. There are languages, however, that directly support this conceptual view, by providing the necessary abstractions. For example, lan-guages like Visual Basic, Visual C++, or Tcl/Tk allow one to separately design the widgets and the code fragments, and to bind events on a widget to the fragments which respond to the event.

Another common event-driven control paradigm is the one based on so-called *triggers*. Triggers became popular in recent years, in conjunction with new developments in the field of so-called active data bases. Since there is no pre-cise and universal definition of a trigger, we will give examples based on a hypothetical language syntax. An *active data base* consists of a conventional underlying (passive) data base and a set of *active rules* (or *triggers*) of the fol-lowing form

```
on event
when condition
do action
```

When the event associated with the rule occurs, we say that the rule is *trig-gered*. A triggered rule is then checked to see if the condition holds. If this is the case, the rule can be executed.

As an example, the following trigger specifies that the total number of employees should be updated as a new employee record is inserted in the data base.

```
on insert in EMPLOYEE
when TRUE
do emp_number ++
```

Concurrent computations

Sometimes it is convenient to structure our software as a set of concurrent units which execute in parallel. This can occur when the program is executed on a computer with multiple CPU’s (multiprocessor). In such a case, if the number of processors coincides with the number of concurrent units, we say that underlying machine that executes the program provides for *physical* parallelism: each unit is in fact executed by its dedicated processor. Parallelism, however, may be simply *logical*. For example, if the underlying machine is a uniprocessor, the logical view of parallel execution may be provided by switching the CPU from one unit to another in such a way that all units appear to progress simultaneously. The switching of the execution of the uniprocessor among the various units can be performed by a software layer, implemented on top of the physical machine, which provides the programmer with a view of an abstract parallel machine where all units are executed simultaneously. The hardware structure might be a multiprocessor, with each processor dedicated to a single unit, or it might be a multiprogrammed uniprocessor.

Concurrency is an important area of computer science, which is often studied in different context: machine architectures, operating systems, distributed systems, databases, etc. In this section we give an overview of how program-ming languages support concurrency. Concurrent programs support concur-rency by allowing a number of units (called *processes*) to execute in parallel (logically or physically).

Processes

A concurrent programming language must provide constructs to define processes. Processes can belong to a type, of which several instances can be cre-ated. The language must define how and by whom a process is initiated, i.e., a new independent execution flow is spawned by an executing unit. It also need to address the issue of process termination, i.e., how can a process be terminated, what happens after a process terminates, etc.

Synchronization and communication

In this section we present some elementary mechanisms for process synchronization and interprocess communication: semaphores , signals and monitors, and rendezvous. Semaphores are low level synchronization mechanisms that are mainly used when interprocess communication occurs via shared variables. Monitors are higher level constructs that define abstract objects used for interprocess communication; synchronization is achieved via signals. Finally, rendevous is another mechanism that combines synchronization and communication via message passing.

Semaphores

A semaphore is a data object that can assume an integer value and can be operated on by the primitives P and V. The semaphore is initialized to a cer-tain integer value when it is declared.

The definitions of P and V are

```
P (s):  if s>0 then s = s - 1
        else suspend current process

V (s): if there is a process suspended on the semaphore
        then wake up process
        else s = s + 1
```

The primitives P and V are assumed to be indivisible, atomic operations; that is, only one process at a time can be executing P or V operations on the same semaphore. This must be guaranteed by the underlying implementation, which should make P and V behave like elementary machine instructions.

Monitors and signals

Concurrent Pascal introduced the signal and monitor constructs into the programming languages. *Signals* are synchronization primitives; monitors describe abstract data types in a concurrent environment. The operations that manipulate the data structure are guaranteed to be executed in mutual exclu-sion by the underlying implementation. Cooperation in accessing the shared data structure must be programmed explicitly by using the monitor signal primitives delay and continue.

```
type fifostorage =
monitor
    var contents: array [1. .n] of integer;  {buffer contents}
        tot: 0. .n; {number of items in buffer}
        in,  {position of item to be added next}
        out: 1. .n; {position of item to be removed next}
        sender, receiver: queue;
    procedure entry append (item: integer);
    begin if tot = n then delay (sender);
        contents [in] := item;
        in := (in mod n)+1;
        tot := tot + 1;
        continue (receiver)
    end;
    procedure entry remove (var item: integer);
    begin if tot = 0 then delay (receiver);
        item := contents[out];
        out := (out mod n) + 1;
        tot := tot - 1;
        continue (sender)
    end;
begin {initialization part}
    tot := 0; in := 1; out := 1
end
```

Producer-consumer example with monitor

Rendezvous

rendezvous concept introduced by the Ada programming language. The construct can be viewed as a high-level mecha-nism that combines synchronization and communication, where communica-tion is based on the *message passing* conceptual paradigm. The construct, per se, can be naturally used to write software for distributed architectures, although its possible interaction with other features in Ada can make this quite difficult. Hereafter we concentrate on the basic properties of rendez-vous; additional features and the interaction with other facilities provided by the language (such as scope rules and exception handling), will be ignored for the sake of simplicity.

```
task Buffer_Handler is      --task declaration
   entry Append (Item: in Integer);
   entry Remove (Item: out Integer);
end;
task body Buffer_Handler is --task implementation
   N: constant Integer := 20;
   Contents: array (1. .N) of Integer;
   In_Index, Out_Index: Integer range 1. .N := 1;
   Tot: Integer range 0. .N := 0;
begin loop
   select
      when Tot < N =>
         accept Append (Item: in Integer) do
            Contents (In_Index) := Item;
         end;
      In_index := (In_Index mod N)+1;
      Tot := Tot+ 1
   or
      when Tot > 0 =>
         accept Remove (Item: out Integer) do
            Item := Contents (Out_Index);
         end;
      Out_Index := (Out_Index mod N) + 1;
      Tot := Tot - 1;
```

```
        end select;
    end loop;
    end Buffer_Handler;
```

FIGURE 59. An Ada task that manages a buffer

Structuring the program

Software design methods

To combat the complexities of programming in the large, we need a systematic design method that guides us in composing a large program out of smaller units— which we call *modules*. A good design is composed of modules that interact with one another in well-defined and controlled ways.

The goal of software design is to find an appropriate modular decomposition of the desired system. Indeed, even though the boundaries between programming in the large and programming in the small cannot be stated rigorously, we may say that programming in the large addresses the problem of modular system decomposition, and programming in the small refers to the production of individual modules. A well-known approach is *informa-tion hiding* which uses the distribution of " secrets" as the basis for modular decomposition.

If a design is composed of highly independent modules, it supports the requirements of large programs:

- Independent modules form the basis of work assignment to individual team members. The more independent the modules are, the more independently the team members can proceed in their work.
- The correctness of the entire system may be based on the correctness of the individual modules. The more independent the modules are, the more easily the correctness of the individual modules may be established.
- Defects in the system may be repaired and, in general, the system may be enhanced more easily because modifications may be isolated to individual modules.

Concepts in support of modularity

A good module represents a useful abstraction; it interacts with other modules in well-defined and regular ways; it may be understood, designed, implemented, compiled, and enhanced with access to only the spec-ification (not the implementation secrets) of other modules. Programming languages provide facilities for building programs in terms of constituent modules.

Procedures and functions are units for structuring small programs, perhaps limited to a single file. Sometimes, we may want to organize a set of related functions and procedures together as a unit. For example, we saw in Chapter 3 how the class construct of C++ lets us group together a data structure and related operations. Ada and Modula-2 provide other constructs for this pur-pose. Before we delve into specific language facilities, we will first look at some of the underlying concepts of modularity. These concepts help motivate the need for the language facilities and help us compare the different language approaches.

Encapsulation

A program unit provides a service that may be used by other parts of the program, called the *clients* of the service. The unit is said to *encapsulate* the service. The purpose of encapsulation is to group together the program components that combine to provide a service and to make only the relevant aspects visible to clients. *Information hiding* is a design method that emphasizes the importance of concealing information as the basis for modularization. Encapsulation mechanisms are linguistic constructs that support the implementation of information hiding modules.

For example, assume that a program unit implements a dictionary data structure that other units may use to store and retrieve <name, “ id” > pairs. This dictionary unit makes available to its clients operations for: inserting a pair, such as <“ Mehdi” , 46>, retrieving elements by supplying the string component of a pair, and deleting elements by supplying the string component of a pair. The unit uses other helper routines and data structures to implement its service. The purpose of encapsulation is to ensure that the internal structure of the dictionary unit is *hidden* from the clients. By making visible to clients only those parts of the dictionary unit that they need to know, we achieve two important properties.

- The client is simplified: clients do not need to know how the unit works in order to be able to use it; and
- The service implementation is independent of clients and may be modified without affecting the clients.

Different languages provide different encapsulation facilities. For example, in C, a file is the unit of encapsulation. Typically, the entities declared at the head of a file are visible to the functions in that file and are also made avail-able to functions in other files if those functions choose to declare them. The declaration:

```
extern int max;
```

states that the variable max to be used here, is defined and storage for it allocated— elsewhere. Variables declared in a C function are local and known only to that function. Variables declared outside of functions are assumed to be available to other units, if they declare them using the extern specifier.

Interface and implementation

A module encapsulates a set of entities and provides access to some of those entities. The available entities are said to be *exported* by the module. Each of the exported entities is available through an interface. The collection of the interfaces of the exported entities form the *module interface*. Clients request the services provided by a module using the module's *interface*, which describes the module's specification. The idea is that the interface is all that the client needs to know about the provider's unit. The implementation of the unit is hidden from the client. The separation of the interface from the implementation contributes to the independence of the client and the server from one another.

A service provider *exports* a set of entities to its clients. A client module *imports* those entities to be able to use the services of the provider module. The exported entities comprise the service provided by the module. Languages also differ with respect to the kinds of entities they allow to be exported. For example, some languages allow a type to be exported and others do not.

The simplest interface, one that proce-dure or function interface. A function declaration such as:

```
int max (int& x, int& y)
```

In C++, where the unit of encapsulation is a class, the interface to the class consists of the interfaces of all the member functions of the class that are available to clients as well as any other entities, such as types and variables, that are made public by the unit.

Ada treats the separation of interface and implementation quite strictly. In Ada, the unit of encapsulation is a package. A package encapsulates a set of entities such as procedures, functions, variables, and types. The package interface consists of the interfaces provided by each of those entities. The Ada package supports encapsulation by requiring the interface of a package (called **package specification**) to be declared separately from the implementation of the package (called **package body**). Figure shows the Ada package specification for our dictionary unit. The implementation of the dictionary package is shown in another Figure. The package body contains all the implementation details that are hidden from the clients. This separation helps achieve both of the goals stated for encapsulation. The package body, as can be seen in the figure, defines both the implementation of the entities defined in the package interface and the implementation of other entities internal to the module. These entities are completely hidden from the clients of the package. The package specification and the package body may appear in different files and need not even be compiled together. To write and compile a client module, only the service's package specification is necessary.

There are significant differences between the packages of Ada and classes of C++. Even from this simple example we can see a difference between the models supported by C++ and Ada. In C++, the client can declare several instances of the dictionary class. In Ada, on the other hand, a module may be declared once only and the client obtains access to only a single dictionary.

```
package Dictionary is
    procedure insert (C:String; I: Integer);
    function lookup(C:String): Integer;
    procedure remove (C: String);
end Dictionary;
```

Figure: Package specification in Ada

```
    package body Dictionary is
    type node;
    type node_ptr is access node;
    type node is
        record
            name: String;
            id: Integer;
            next: node_ptr;
        end record;
    root: node_ptr;
    procedure insert (C:String; I: Integer) is
    begin
        --imlementation...
    end insert;
    function lookup(C:String): Integer is
    begin
        --imlementation...
    end lookup;
    procedure remove (C: String) is
    begin
        --imlementation...
    end remove;
begin
    root := null;
end Dictionary;
```

FIGURE. Package body in Ada

Separate and independent compilation

The idea of modularity is to enable the construction of large programs out of smaller parts that are developed independently. At the implementation level, independent development of modules implies that they may be compiled and tested individually, independently of the rest of the program. This is referred to as *independent* compilation. The term *separate* compilation is used to refer to the ability to compile units individually but subject to certain ordering con-straints. For example, C supports independent compilation and Ada supports separate compilation. In Ada, as we will see later, some units may not be compiled until other units have been compiled. The ordering is imposed to allow checking of inter-unit references. With independent compilation, nor-mally there is no static checking of entities imported by a module.

To illustrate this point, consider the program sketch in Figure, written in a hypothetical programming language. Separate compilation means that unit B,which imports routine X from unit A, must be compiled after A. This allows any call to X issued by B to be checked statically against X's definition in A. If the language allows module interfaces to be compiled separately from their bodies, only A's interface must be compiled before B; its body can be com-piled at any time after its corresponding interface has been compiled.

```
Unit A                                  Unit B

   export routine X (int, int);            . . .

                                           call X (. .
   . . .                                   .);

end A                                      . . .

                                        end B
```

FIGURE. Sketch of a program composed of two units

Libraries of modules

We have seen that C++ class and Ada's package make it possible to group related entities into a single unit. But large programs consist of hundreds or even thousands of such units. To control the complexity of dealing with the large number of entities exported by all these units, it is essential to be able to organize these units into related groups. For example, it is difficult to ensure that all the thousands of units have unique names! In general, we can always find groupings of units that are related rather closely.

Language features for programming in the large

All programming languages provide features for decomposing programs into smaller and largely autonomous units. We refer to such units as *physical* modules; we will use the term *logical* module to denote a module identified at the design stage. A logical module represents an abstraction identified at the design stage by the designer. A logical module may be implemented by one or more physical modules. The closer the relationship between the physical modules and logical modules is, the better the physical program organization reflects the logi-cal design structure.

We will discuss Pascal, C, C++, Ada, and ML. Pascal and C are viewed here as a representative of the class of traditional, minimalist, procedural lan-guages. Our conclusions about them hold, with minor changes, for other members of the class such as FORTRAN. C++ is a representative of class-based languages. Ada is a representative of module-based languages, although the 1995 version of the language has enhanced its object-orientation support. ML is reviewed as a representative of functional languages.

Pascal

The only features provided by Pascal for decomposing a program into mod-ules are procedures and functions, which can be used to implement proce-dural abstractions. The language thus only supports procedural programming. Some later versions of the language have modified the original version of Pascal extensively by adding object-oriented programming features.

A Pascal program has the following structure.

program program_name (files);

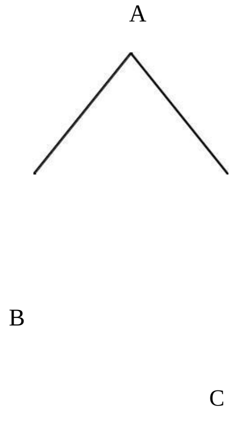
declarations of constants, types, variables, procedures and functions;

begin

  statements (no declarations)

end.

A program consists of declarations and operations. The operations are either the built-in ones provided by the language or those declared as functions and procedures. A procedure or function itself may contain the declaration of con-stants, types, variables, and other procedures and functions.



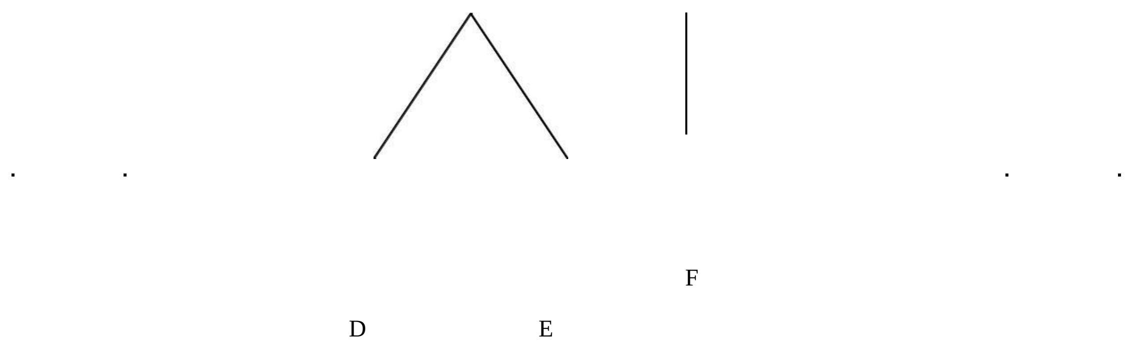


FIGURE . Static nesting tree of a hypothetical Pascal program

To assess the structure of Pascal programs, consider the following example. Suppose that the top-down modular design of a module A identifies two mod-ules B and C providing subsidiary procedural abstractions. Similarly, module B invokes two private procedural abstractions provided by modules D and E. Module C invokes a private procedural abstraction provided by F. Figure shows a nesting structure for a program that satisfies the design constraints.

C

C provides functions to decompose a program into procedural abstractions. In addition, it relies on a minimum of language features and a number of con-ventions to support programming in the large. These conventions are well recognized by C programmers and are even reflected in tools that have been developed to support the language. Indeed, a major portion of the programming in the large support is provided by the file-inclusion commands of the C preprocessor. Thus, even though the compiler does not provide any explicit support or checking for inter-module interaction, the combination of conventions and the preprocessor has proven in practice to be an adequate and popular way to support programming in the large.

Figure shows the header and implementation files for a module providing a stack data structure. language provides no encapsulation facilities. For example, the main program in Figure 70 has complete access to the internal structure of the stacks s1 and s2. In fact, this property is used by the main pro-gram to initialize the stacks s1 and s2 to set their stack pointers (top) to 0. There are ways to implement this program to reduce this interference between client and server but all depend on the care taken by the programmer. There is no control over what is exported: by default, all entities in a file are exported. Files may be compiled separately and inter-file references are resolved at link time with no type-checking. A file may be compiled as long as all the files it includes are available.

```
/* file stack.h */

/*declarations exported to clients*/
```

```
typedef struct stack {
    int elments[100]; /* stack of 100 ints */
    int top;            /*number of elements*/
};
extern void push(stack, int);
extern int pop(stack);
/* end of file stack.h */
/*****----------------------end of file          ****/


/*file stack.c */
/*implementation of stack operations*/
#include"stack.h"
void push(stack s, int i) {
    s.elements[s.top++] = i;
};
int pop (stack s) {
    return --s.top;
};
                                                 ****/
/*****----------------------end of file


/*file main.c */
/*A client of stack*/
#include "stack.h"
void main(){
    stack s1, s2;         /*declare two stacks */
    s1.top = 0; s2.top = 0; /* initialize them */
    int i;
    push (s1, 5); /* push something on first stack */
    push (s2, 6); /* push something on second stack*/
    ...
    i = pop(s1); /* pop first stack */
    ...
}
```

FIGURE. Separate files implementing and using a stack in C

C++

C++ is based on C and it shares C's reliance on conventions and unit of phys-ical modularity as the file. As C, C++ provides functions as a decomposition construct to implement abstract operations. Nevertheless, C++'s most important enhancements to C are in the area of programming in the large. In particular, the class construct of C++ provides a unit of logical modularity that supports the implementation of information hiding modules and abstract data types. Combined with templates, classes may be used to implement generic abstract data types. The class provides encapsulation and control over inter-faces. In this chapter, we review the use of classes as modules.

Encapsulation in C++

The unit of logical modularity in C++ is the class. A class serves several pur-poses including:

- A class defines a new (user-defined) data type.
- A class defines an encapsulated unit.

Entities defined by a class are either *public*— exported to clients—o r *pri-vate*—h idden from clients. Classes may be nested. But as we saw in the case of Pascal, nesting may be used only for programming in the small and is of limited utility for program-ming in the large.

Program organization

Classes define the abstractions from which the program is to be composed. The main program or a client creates instances of the classes and calls on them to perform the desired task. We saw the definition of a C++ template module implementing a generic abstract data type stack. Figure shows a class implementing a stack of integers. The implementation separates the interface and the implementation in different files.

```
/* file stack.H */

/*declarations exported to clients*/

class stack {

public:

    stack();

    void push(int);

    int push pop();

private:

    int elments[100]; /* stack represented as array */

    int top = 0;      /*number of elements*/

};

// the implementation follows and may be in a
separate file void stack::push( int i) {

    elements[top++] = i;

};

int stack::pop (int i) {
    return elements[--top];

};

/*end of stack.H*/


/*main.c */

/*A client of stack*/

#include “ stack.h”

main(){

    stack s1, s2;  /*declare two stacks */

    int i;

    s1.push (5); /* push something on first stack */

    s2.push (6); /* push something on second stack*/

    ...
```

```
     i = s1.pop();/* pop first stack */
... ...
}
```

FIGURE . Stack class in C++

Ada

Ada was designed specifically to support programming in the large. It has elaborate facilities for the support of modules, encapsulation, and interfaces. Rather than relying on convention as in C and C++, Ada makes an explicit distinction between specification and implementation of a module. A file may be compiled if the specifications of the modules it uses are available. Thus, Ada naturally supports a software development process in which module specifications are developed first and implementation of those modules may proceed independently. Ada also requires the existence of a compile-time library in which module specifications are compiled. A module may be com-piled if all the module specifications it needs are already in the library.

Encapsulation in Ada

The package is Ada's unit of modularity. An Ada module encapsulates a group of entities and thus supports module-based programming. We have already seen that the language's explicit distinction between module specifi-cation and module body forces the programmer to separate what is expected by the module from what is hidden within the module. Additionally, Ada sup-ports concurrent modules or tasks.

Program organization

An Ada program is a linear collection of modules that can be either subpro-grams or packages. These modules are called units. One particular unit that implements a subprogram is the main program in the usual sense. Module declarations may be nested. Consequently, a unit can be organized as a tree structure of modules. Any abuse of nesting within a unit causes the same problems discussed for Pascal. These problems can be mitigated by the use of the subunit facility offered by the language. This facility permits the body of a module embedded in the declarative part of a unit (or subunit) to be written separately from the enclosing unit (or subunit). Instead of the entire module, only a *stub* need appear in the declarative part of the enclosing unit. The fol-lowing example illustrates the concept of the subunit.

```
procedure X ( ... ) is                          --unit specification
```

```
W: INTEGER;
package Y is                              --inner unit specification
   A: INTEGER;
   function B (C: INTEGER) return INTEGER;
end Y;
package body Y is separate;               --this is a stub
```

```
begin        -- uses of  package Y and variable W
    .....
    ...
end X;
-----------------------------------next file--------------

separate (X)
package body Y is
    procedure Z (...) is separate;                    --this is a stub
    function B (C: INTEGER) return INTEGER is
    begin      --use procedure Z
        ...
        ...
        ...
    end B;
end Y;
-----------------------------------next file--------------

separate (X.Y)

            procedure Z (...) is
            begin
                ...
            end Z;
```

The interface of an Ada unit consists of the **with** statement, which lists the names of units from which entities are imported, and the *unit specification* (enclosed within a **is... end** pair), which lists the entities exported by the unit. Each logical module discovered at the design stage can be implemented as a unit. If the top-down design was done carefully, logical modules should be relatively simple. Consequently, the nesting within units should be shallow or even nonexistent. Ada does not forbid an abuse of nesting within units. Actu-ally, the entire program could be designed as a single unit with a deeply nested tree structure. It is up to the designer and programmer to achieve a more desirable program structure.

```
with X;

package T is

    C: INTEGER;
    procedure D (...);

end T;

package body T is

    ...

    ...

    ...

end T;
```

Similarly, the following procedure U can legally call procedure T.D and access variable T.C. On the other hand, unit X is not visible by U.

```
with T;

procedure U (...) is

    ...

end U;
```

Interface and implementation

We have already seen in Section that Ada strictly separates the specification and body of a package. In the previous section, we have seen how the **use** and **with** clauses are used to import services from packages. These facilities are used also to support separate compilation.

ML

Modularity is not only the province of imperative languages. The notion of module is important in any language that is to be used for programming in the large. For example, ML is a functional programming language with extensive support for modularity and abstraction.

Encapsulation in ML

A module is a separately compilable unit. A unit may contain structures, sig-natures, and functions. Structures are the main building blocks; signatures are used to define interfaces for structures; functors are used to build a new struc-ture out of an existing structure.

The ML *structure* is somewhat like the Ada package, used to group together a set of entities. For example, our dictionary example package of Figure may be written in ML as given in Figure

```
structure Dictionary =
struct
    exception NotFound;

    val root = nil;  (*create an empty dictionary*)

 (* insert (c, i, D) inserts pair <c,i> in dictionary D*)
    fun insert (c:string, i:int, nil) = [(c,i)]
    |    insert (c, i, (cc, ii)::cs) =
         if c=cc then (c,i)::cs
         else (cc, ii)::insert(c,i,cs);

 (* lookup (c, D) finds the value i such that pair <c,i> is in dictionary D *)
    fun lookup(c:string, nil) = raise NotFound
    |    lookup (c, (cc,ii:int)::cs) =
         if c = cc then ii
         else lookup(c,cs);
end;
```

FIGURE. Dictionary module in ML (types string and int are not necessary but used for explanation here)

Such a structure definition corresponds to the package body in that it gives the implementation for the entities being defined. It also has the property that all the entities are exported. This structure exports an exception, NotFound, a variable root, and two functions insert and lookup. To use the structure, a client uses the dot notation:

```
val D = Dictionary.create;  (*create an empty dictionary *)
val newD = Dictionary.insert (“ Mehdi” , 46, D);  (*insert a pair*)
```

```
...
D.lookup(“ Mehdi” , D); (*produces value 46*)
```

Interface and implementation

The signature of a structure definition consists of the signatures and types of all the entities defined in the structure. ML also provides a construct to define a signature independently of any structure. A signature may be viewed as a specification for a module. For example, Figure gives the signature of a module that exports an exception called NotFound and a function called lookup. A signature may be used as a specification for a structure. For exam-ple, we may use the signature of Figure 76 to restrict the exported entities of the structure of Figure. The system will do type checking to ensure that the structure provides at least what the signature requires.

```
signature DictLookupSig = sig
   exception NotFound;

   val lookup : string * (string * int) list -> int
end
```

FIGURE. A signature defintion for specialized dictionary

We can use the structure and signature we have to create a new module with a restricted interface and use it accordingly:

```
structure LookupDict: DictLookupSig = Dictionary;

val L = LookupDict.create; (* not allowed, must be done by someone else using
a different interface *)
lookupDict.lookup(“ Mehdi” , L);
lookupDict.insert(“ Carlo” , 50, L); --error, insert not available
```

ML also supports the concept of *generic* modules or structures. The signature facility may be combined with generic structures to instantiate a structure for particular types. For example, the dictionaries that we have defined so far, both in Ada and in ML have been specific to <string, integer> pairs.

Generic units

One important approach to developing large pro-grams is to build them from existing modules. Traditional software libraries offer examples of how such existing modules may be packaged and used. One of the criteria for evaluating the suitability of a language for programming in the large is whether it provides language mechanisms that enable the construction of independent components, the packaging together of related components, the use and combination of multiple libraries by clients, etc. We have seen the namespaces of C++ and the libraries and child libraries of Ada 95 as language mechanisms explicitly developed for the support of such packaging of related and independent software components. In this section, we concentrate on genericity as a mechanism for building individual modules that are general and thus usable in many contexts by many clients.

Unit 4

Object-Oriented Languages

Concepts of object-oriented programming

There are several definitions of what object-oriented programming is. Many people refer to an object-oriented program as any program that deals with entities that may be informally called “objects.” In contrast to traditional procedural languages in which the programs consist of procedures and data structures, objects are entities that encapsulate data and related operations. For example, given a stack data structure s, in a procedural language we would call a push operation to add an element as in:

```
push (s, x);
```

Dealing with objects, as we have seen in C++, we tell the stack object to push an element onto itself, as in:

```
s.push(x);
```

We have seen that in C++ and Eiffel we can use classes to define and create objects. We call a programming language that supports the definition and use of objects *object-based*. *Object-oriented* programming languages support additional features. In particular, object-oriented programming languages are characterized by their support of four facilities:

- abstract data type definitions,
- inheritance,
- inclusion polymorphism, and
- dynamic binding of function calls.

The pure terminology of object-oriented languages refers to *objects* that are *instances* of *classes*. An object contains a number of *instance variables* and supports a number of *methods*. A *message* is sent to an object to request the invocation of one of its methods. For example, s.push(5) is interpreted as send-ing the message push(5) to object s. The message is a request to s to invoke its method push with the parameter 5. The syntax of Smalltalk reflects this inter-pretation directly. In Smalltalk, the same statement would be written as: 5. In a dynamically-typed language such as Smalltalk, the object that receives a message first checks to see that it can perform the method requested. If not, it reports an error. Other languages, such as Eiffel and C++, allow polymor-phism but restrict it to enable static type checking. Thus, such languages combine polymorphism with strong typing.

Classes of objects

The first requirement for object-oriented programming is to be able to define abstract data types. We have already seen that this can be done using the class construct. As an example, recall the following definition of a stack class

```
class stack{
    public:
        void push(int) {elements[top++] = i;};
        int pop() {return elements[--top];};
    private:
        int elements[100];
        int top=0;
};
```

This class is a particular implementation of a fixed-size stack abstraction. We may use objects of this class in many different applications. As observed in Chapters 3 and 5, the class construct enables us to encapsulate the representa-tion used for the data structure and export useful operations to be used by cli-ents.

A client may create as many objects of the stack class as desired:

```
stack s1, s2;
s1.push(3);
s1.push(4);
s2.push(3);
if (s1.pop() == s2.pop) {...}
```

Clients may create objects of this class just as they may create variables of language-defined types. In fact, classes are user-defined types. They share most properties of language-defined types, including storage properties. For example, the above fragment creates stacks s1 and s2 as automatic variables. We may also create stacks in the free store:

```
stack* sp = new stack
```

To access member functions (e.g., pop) the following notations denote equivalent expressions:

```
(*sp).pop() ;   and (more commonly used)

sp -> pop();
```

While useful, the class facility only addresses the question of how to encapsulate useful abstractions. What we have seen so far does not address the need to create new abstractions based on existing ones. Inheritance is the mechanism used for this purpose.

Inheritance

Inheritance is a linguistic mechanism that allows us to do just that by defining a new class which "inherits" the properties of a parent class. We may then add new properties to the child class or redefine inherited properties. The terminology in C++ is to *derive* a class from a *base* class. In this case, we want to derive a counting_stack from stack as shown below:

```
class counting_stack: public stack {

    public:

      int size(); //return number of elements on the stack

};
```

This new class simply inherits all the functions of the class stack[1]. All public member functions of stack become public member functions of (that's what the public before stack specifies). We have also specified that there will be a new public function size() which is intended to return the number of elements stored in the stack. The class stack is called a base class or the parent class of counting_stack. The class counting_stack is said to be derived from its base class. The terms *subclass* and *superclass* are also used to refer to derived and base classes respectively.

Polymorphism

All classes derived from the same base class may be viewed informally as specialized versions of that base class. Object-oriented lan-guages provide polymorphic variables that may refer to objects of different classes. Object-oriented languages that adopt a strong type system limit the polymorphism of such variables: usually, a variable of class T is allowed to refer to objects of type T or any classes derived from T.

In the case of our example stack and counting_stack in the previous section, this means that a variable of type stack may also refer to an object of type counting_stack. In purely object-oriented languages such as Eiffel and Small-talk, *all* objects are referred to through references and references may be polymorphic. In C++, only pointer, reference variables, and by reference parameters may be polymorphic. That is, a stack pointer may also point to a counting_stack object. Similarly, a formal by reference parameter expecting an actual parameter of type stack can refer to an actual parameter of type counting_stack. As an example, if we have two pointer variables declared:

```
stack* sp = new stack;

counting_stack* csp = new counting_stack;

...

sp = csp;  //okay

...

csp = sp; //statically not sure if okay--disallowed
```

The assignment sp = csp; allows a pointer to a class to point to an object of a subclass. That is, the type of the object that is currently pointed at by sp can be of any of its derived types such as counting_stack. The assignment csp = sp; is not allowed in C++ because C++ has a strong type system. If this assignment were allowed, then a later call to csp->size() would be statically valid but may lead to a runtime error because the object pointed at by sp may not support the member function size(). A language with a weak type system, such as Small-talk, would allow such an assignment and defer error checking to runtime.

In C++, if we do not use pointers, then the situation is different:

```
stack s;

counting_stack cs;

...

s = cs; //okay, object is coerced to a stack (no more size operation available)

cs = s; //not allowed because a later cs.size() would look syntactically okay but
not work at runtime
```

The assignment s = cs; is legal and is an example of ad hoc polymorphism. In fact, what happens is that the values stored in object cs are copied into the cor-responding elements of s. This kind of coercion, of course, loses some of the values stored in cs, just as coercion from float to int loses some information.

Dynamic binding of calls to member functions

A derived class may not only add to the functionality of its base class, it may also add new private data and *redefine* or *override* some of the operations provided in the base class. For example, in we may decide to provide a new implementation of the push function because we may want to keep track of the number of times push has been called. Now, if sp is a reference to a stack variable and csp is a reference to a counting_stack variable, we expect that csp->push() will call the push of a counting_stack object but what about sp->push()? Since sp is a polymorphic variable, it may be pointing at a counting_stack object or a stack object. This raises an issue of proper binding of operations. Consider the following example:

```
stack* sp = new stack;
counting_stack* csp = new counting_stack;
...
sp.push();  // stack::push
csp.push(); // counting_stack::push
...
sp = csp;  //assignment is okay
...
sp.push();  //which push?
```

Which is the push operation invoked in the last statement, stack::push or counting_stack::push? Because the assignment sp = csp is allowed, at run-time sp may be pointing to a stack object or to a counting_stack object. Should the choice of which routine to call be made statically, in which case stack::push() will be called, or dynamically, in which case counting_stack::push() will be called. So-called purely object-oriented languages, such as Smalltalk and Eiffel, bind the choice dynamically based on the type of the object. In fact, as stated, dynamic binding (often called *dynamic dispatching* in object-oriented terminology) is one of the tenets for object-oriented programming languages. C++, however, not being a purely object-oriented language, provides features for both static and dynamic binding. Section 6.3.1 presents the C++-specific features.

Inheritance and the type system

In the previous section, we have described the basic components of object-oriented programming languages. The interaction between inheritance and type consistency rules of the language raises a number of interesting issues. In this section, we consider some of these issues.

Subclasses versus subtypes

The concept of subtype with which we defined a new type as a subrange of an existing type. For example, we defined week_day as a subrange of day. Subtyping introduces a relationship among objects of the subtype and objects of the the parent type such that objects of a subtype may also be viewed as objects of the parent type. For example, a week_day is also a day. This relationship is referred to as the *is-a* relationship: week_day *is-a* day. The subtype relationship is generalizable to user-defind types such as those defined by classes. For example, a counting_stack *is-a* stack but not vice versa.

But not all subclasses create subtypes. If a derived class only adds member variables and functions or redefines existing functions in a *compatible* way, then the derived class defines a subtype. If it hides some of the parent's member variables and functions or modifies them in an incompatible way, then it does not create a subtype. Therefore, whether a derived class defines a subtype depends on the definition of the derived class and is not guaranteed by the language.

What does it mean for a function f in a derived class to override a function f in a base class in a *compatible* way? Informally, it means that an occurrence of base::f(x) may be replaced by derived::f(x) without risking any type errors. For example, if the signature of the function derived::f is identical to the signature of base::f, no type errors will be introduced as a result of the replacement. We will examine this issue more deeply in the next subsection.

Strong typing and polymorphism

Strong type systems have the advantage of enabling type errors to be caught at compile-time. Statically typed languages provide a strong type sys-tem. In this section we discuss how object oriented languages can rely on dynamic dispatch as a fundamental principle and yet adopt a strong type system.

Let us assume that we have a base class base and a derived class derived and two objects derived from them:

```
class base {...};

class derived: public base {...};

...
```

```
base* b;

derived* d;
```

We have seen that we may assign d to b but not b to d. The question is under what circumstances can we guarantee that an assignment

```
b = d;
```

will not lead to a type violation at runtime? We may ask this question in terms of *substitutability*: can an object of type derived be substituted for an object of class base in every context? Or in general, can an object of a derived type be substituted for an object of its parent type in every context, and is such a kind of polymorphism compatible with strong typing? If substitutability is ensured, the derived type can be viewed as a subtype of the parent type. To answer this question we need to examine the contexts under which objects are used. By imposing some restrictions on the use of inheritance, the language can ensure substitutability. Below we examine several sufficient (but not nec-essarily necessary) restrictions.

Type extension

If the derived class is only allowed to extend the type of the parent class, then substitutability is guaranteed. That is, if derived does not modify any member functions of base and does not hide any of them, then it is guaranteed that any call b.f(...) will be valid whether b is holding a base object or a derived object. The compiler may do type-checking of any uses of base variables solely based on the knowledge that they are of type base. Therefore static type check-ing can guarantee the lack of runtime violations. Type extension is one of the mechanisms adopted in Ada 95.

Overriding of member functions

Restricting derived classes to only extend the parent type is a severe limita-tion on an object-oriented language. In fact, it rules out dynamic dispatch completely. Many languages allow a derived class to redefine an inherited member function. For example, as in our previous example, we may derive a square class from a polygon class. The square class may redefine the general perimeter function from the base class by a more efficient version of a function. In C++, the base class must specify the function perimeter as a vir-tual function, giving derived classes the opportunity to override its definition. That is,

```
class polygon {

   public:

      polygon (...) {...} //constructor

      virtual float perimeter () {...};
```

```
        ...
    };
    class square: public polygon {
      public:
        ...
        float perimeter() {...};  //overrides the definition of perimeter in polygon
    };
```

Inheritance hierarchies

Hierarchical organization is an effective method of controlling complexity. The inheritance relationship imposes a hierarchy and provides a mechanism for the development of a hierarchically organized families of classes. In this section we discuss several issues raised by inheritance hierarchies.

Single and multiple inheritance

In Simula 67, Ada, and Smalltalk, a new class definition is restricted to have only one base class: a class has at most one parent class. These languages have a *single-inheritance* model.

C++ and Eiffel have extended the notion of inheritance to allow a child to be derived from more than one base class. This is called *multiple inheritance*. For example, if we have a class displayable and a class polygon, we might inherit from both to define a displayable rectangle:

```
    class rectangle: public displayable, public polygon {
    ...
    }
```

The introduction of multiple inheritance into a language raises several issues. For example, there may be name clashes between parents. For example, a dis-playable class and a bank_account class may both provide a member function draw() and inheriting from both to build a displayable bank_account may cause problems. In this case, which draw() function is exported by the derived class? The derived class needs a way to both access and to export the desired mem-bers.

The successful use of multiple inheritance requires not only well-designed inheritance hierarchies but also orthogonally designed classes that may be combined without clashing. In practice, the use of multiple inheritance requires much care. Whether its benefits outweigh its complexity is an open question. Java, which adopts many features of C++, uses only single inherit-ance but introduce separate interfaces and supports the idea of inheriting from multiple interfaces.

Implementation and interface inheritance

One of the promises of object-oriented programming is that new software components may be constructed from existing software components. This would be a significant contribution to programming in the large issues. To what extent does inheritance support a methodology for such incremental building of components?

From a software engineering view, interface inheritance is the right method-ology but to rely only on interface inheritance requires both a well-designed base class and efficient language implementations. A well-designed inherit-ance hierarchy is a requirement for the successful use of object-oriented pro-gramming in any case. Any hierarchy implies that the nodes closer to the root of the hierarchy affect a larger number of the leaf nodes of the hierarchy. If a node close to the root needs to be modified, all of its children are affected. As a result, even though inheritance supports the incremental creation of soft-ware components, it also creates a tightly-dependent set of components. Mod-ifications of base classes may have far reaching impact.

Object-oriented programming support in programming anguages

The way different languages support object-oriented programming is related to the philosophy of the language and more specifically to the language's object and encapsulation models. In C++, the class construct defines a user-defined type. Object-oriented features have been added to the language to allow programmers who want to use object-oriented programming to do so. In Eiffel, the class construct defines an abstract data type. The language has been designed to support the object-oriented programming style exclusively. In Ada 95, the package is simply an encapsulation mechanism for packaging a set of related entities. It is neither necessarily a type, nor an abstract data type. It may be used to support those notions, however. Ada 95 has some object-orientation features but the language remains a module-oriented language. In this section, we examine these languages a little more closely from an object-oriented view.

C++

C++ supports object-oriented programming by providing classes for abstract data types, derived classes for inheritance, and virtual functions for dynamic binding. This support is provided with static type checking.

The language supports both inheritance and multiple inheritance for defining new classes.

Classes

We have already seen the use of C++ classes as a definition mechanism for abstract data types. C++ provides the programmer with control over the creation, initialization, and cleanup of objects of such abstract data types. In par-ticular, one or more constructors may be defined for a class. Such constructors are invoked automatically to initialize objects of the class at cre-ation time. A constructor has the same name as the class. By analogy to a con-structor, a *destructor* is invoked automatically when the object is destroyed— either explicily through a call to delete or implicitly when the object goes out of scope. The ability to control what happens when an object is destroyed is critical for complex data types that may have allocated substructures in the heap. For example, simply deleting a pointer to the head of a list may leave the entire list inaccessible in the free store. A destructor gives the progammer the possibility to clean up after the object properly based on the requirements of the object. The name of a destructor is the same as the class name preceded by ~ (i.e. the complement of the constructor).

Virtual functions and dynamic binding

By default, a function call is bound to a function definition statically. If the programmer wants a function to be selected dynamically, the function must be declared as virtual in the base class and then redefined in any derived classes. For example, suppose we define a class student that supports some operations including a print() operation. We expect to derive different types of students from this class, such as college_student and graduate_student. First we define the student class and define a default print() function:

```
class student{
public:
...
virtual void print(){...};
};
```

The virtual qualifier on print() says that classes derived from student may rede-fine the function print(). If they do, then the appropriate print() function will be selected dynamically. For example:

```
class college_student: public student{
    void print() {
    . . . // specific print for college_student
            }
        };
```

defines a derived class that inherits all its operations from student but supplies a new definition for print(). Now, the following code sequence shows the effect of dynamic binding:

```
student* s;
college_student* cs;
...
s->print(); //calls student::print()
s = cs; // okay
s->print(); //calls college_student::print()
```

Remember that in C++, the binding is normally static. The virtual function and pointers are used to effect dynamic binding.

Overloading, polymorphism, and genericity

the use of ad-hoc polymorphism in the support of overloading of operators and functions, in which case the binding of the operator or function is done at compile-time based on the types of the arguments. If a derived class redefines a virtual function f of a base class b, the base class defines a polymorphic type and the function call b.f() is a call to a polymorphic function resolved dynamically on the basis of the type of the object referred to by b. Since the object must belong to a sub-class of the class to which b is declared to point, this is a case of inclusion polymorphism. If the function is not declared to be virtual in the base class, then the two functions in the base and derived classes are treated as simply overloading the same function name: the proper function to call is selected at compile-time. Finally, we have seen that C++ also supports generic functions. If a function f(a,b) is generic, the types of a and b are used at compile time to instantiate a function that matches the types of the parameters a and b. There is no dynamic dispatch in this case.

Ada 95

The original version of the Ada language, introduced in 1983, was an object-based language. The package construct may be used to create objects that encapsulate both data, possibly private, and related operations. This enabled object-based programming. Since the introduction of Ada, however, the con-cepts of object-oriented programming have become better understood. As a result, a number of features were added to Ada 95 to support object-oriented programming techniques. In particular, **tagged types** support derivation of a new type from an existing type, and a technique called " classwide program-ming" c ombined with tagged types supports dynamic dispatch. Another fea-ture of Ada 95 is the ability to define abstract types which in turn enable the association of multiple implementations with the same interface. Such flexi-bility is usually associated with object-oriented programming. We discuss these features below.

Tagged type

With Ada 95, a type declaration may be designated as *tagged*, in which case it is possible to derive new types from it by extending it. This facility allows us to define a hierarchy of related types. For example, Figure defines a tagged type named as having certain set of basic properties such as X and Y coordinates of its center and three functions: one to compute its Distance from the origin, another to Move it to a new position, and another to Draw it as a predefined icon on the screen. We might then derive other objects such as *points* and *circles* which each extend the basic object in their own ways.

```
package Planar_Objects is
    type Planar_Object is tagged
    record
        X: Float := 0.0; --default initial value of the center's x coordinate
        Y: Float := 0.0; --default initial value of the center's y
    coordinate end record;
    function Distance (O: Planar_Object) return Float;
    procedure Move (O: inout Planar_Object; X1, X2: Float);
    procedure Draw (O: Planar_Object);
end Planar_Objects;
```

FIGURE An Ada 95 package that defines a tagged type Planar_Object

As a result, the derived types can easily be coerced to the parent type by simply ignoring the additional fields. Thus, the following statements are valid:

```
O1: Planar_Object; -- basic object at origin
O2: Planar_Object (1.0, 1.0); -- on the diagonal
C: Circle := (3.0, 4.5, 6.7);
...
O1 := Planar_Object(C); -- coercion by ignoring the third field of C
```

What if we want to do the assignment in the opposite direction? As opposed to C++, this can be done but since the object on the right hand side does not have all the necessary fields, they must be provided by the programmer explicitly. For example:

```
C := (O2 with 4.7);
```

Dynamic dispatch through classwide programming

The tagged types of Ada 95 are also used to support dynamic binding of func-tion calls. The tag is an implicit field of an object and is used at runtime to identify the object's type. For example, suppose that we want to write a pro- cedure that will process a collection of objects that may be Points, Rectangles, or Circles. We need to declare this procedure as accepting a polymorphic type that includes all these types. The 'Class attribute of Ada 95 constructs exactly such a class. That is, the expression T'Class applied to a tagged type T is the union of the type T and all the types derived from T. It is called a *class-wide* type. In our example, we can write our procedure as:

```
procedure Process_Shapes (O: Planar_Object'Class) is
...
begin
   ...
   ... Draw (O) ...;  --dispatch to appropriate Area procedure
   ...
end Process_Shapes;
```

Since it is often useful to access objects through pointers, it is also possible to declare polymorphic pointers such as:

```
type Planar_Object_Ptr is access all Planar_Object'Class;
```

Abstract types and routines

As in C++ and Eiffel, Ada 95 supports the notion of top-down design by allowing tagged types and routines (called subprograms in Ada) to be declared abstractly as a specification to be implemented by derived types. For example, we might have declared our Planar_Object type before as an abstract type as in

```
package Planar_Objects is

    type Planar_Object is abstract tagged null record;

    function Distance (O: Planar_Object'Class) return Float is abstract;

    procedure Move (O: inout Planar_Object'Class; X, Y: Float)  is abstract;

    procedure Draw (O: Planar_Object'Class) is
abstract; end Objects;
```

FIGURE 83. An abstract type definition in Ada

This package will not have a body. It is only a specification. By applying der-ivation to the type Planar_Object, we can build more concrete types. As before, we may derive a tree of related types. Once concrete entities (records and sub-programs) have been defined for all the abstract entities, we have defined objects that may be instantiated.

Eiffel

Eiffel was designed as a strongly-typed object-oriented programming language. It provides classes for defining abstract data types, inheritance and dynamic binding. Classes may be generic based on a type parameter.

Classes and object creation

An Eiffel class is an abstract data type. As we have seen in Chapter 3, it pro-vides a set of **feature**s. As opposed to C++, all Eiffel objects are accessed through a reference. They may not be allocated on the stack. Again in contrast to C++, object creation is an explicit, separate step from object declaration. In one statement, a reference is declared and in a following statement, the object is created. As in:

. .

```
b: BOOK; --declaration of a reference to BOOK
objects !!b; --allocating and initializing an object that
b points to
```

The language provides a predefined way of creating and initializing objects, based on their internally defined data structure. It is also possible to provide user-defined " creator" routines for a class which correspond to constructors of C++.

Inheritance and redefinition

A new class may be defined by inheriting from another class. An inheriting class may **redefine** one or more inherited features. We saw that C++ requires the redefined function to have exactly the same signature as the function it is redefining. Eiffel has a different rule: the signature of the redefining feature must conform to the signature of the redefined feature. This means that for redefined routines, the parameters of the redefining routine must be assign-ment compatible with the parameters of the redefined routine.

Consider the following Eiffel code sequence:

```
class A
feature
   fnc (t1: T1): T0 is
   do
   ...
   end -- fnc
end --class A

class B
inherit
   A redefine fnc
end
```

```
feature

fnc (s1: S1): S0 is

    do

    ...

    end -- fnc

end --class B

...

a: A;
b: B;
...

a := b;

...

... a.fnc (x)...
```

The signature of fnc in B must conform to the signature of fnc of A. This means that S0 and S1 must be assignment compatible with T0 and T1 respectively. The Eiffel **require** and **ensure** clauses which are used to specify pre- and post-conditions for routines and classes may be used as a design tool to impose a stronger discipline on redefinitions.

Smalltalk

Smalltalk was the first purely object-oriented language, developed for a spe-cial purpose machine, and devoted to the devopelopment of applications in a highly interactive single-user personal workstation environment. It is a highly dynamic language, with each object carrying its type at runtime. The type of of a variable is determined by the type of the object it refers to at runtime. Even the syntax of Smalltalk reflects its object orientation.

All objects are derived from the predefined class object. A subclass inherits the instance variables and methods of its parent class (called its superclass) and may add instance varaiables and methods or redefine existing methods. A call to an object is bound dynamically by first searching for the method in the subclass for the method. If not found, the search continues in the superclass and so on up the chain until either the method is found or the superclass object is reached and no method is found. An error is reported in this case.

A class defines both *instance* variables and methods, and *class* variables and methods. There is a single instance of the class variables and methods avail-able to all objects of the class. In contrast, a copy of each instance variable and method is created for each instance of an object of the class.

Unit 5

Functional programming languages

Functional programs reduce the impact of side-effects further, or even eliminate them entirely, by relying on mathematical functions, which operate on *values* and produce *values* and have no side-effects.

Characteristics of imperative languages

Imperative languages are characterized by three concepts: variables, assignment, and sequencing. The state of an imperative program is maintained in program variables. These variables are associated with memory locations and hold values and have addresses. We may access the value of a variable either through its name (directly) or through its address (indirectly). The value of a variable is modified using an assignment statement. The assignment statement introduces an order-dependency into the program: the value of a variable is different before and after an assignment statement. Therefore, the meaning (effect) of a program depends on the order in which the statements are written and executed. While this is natural if we think of a program being executed by a computer with a program counter, it is quite unnatural if we think of mathematical functions. In mathematics, variables are bound to values and once bound, they do not change value. Therefore, the value of a function does not depend on the order of execution. Indeed, a mathematical function defines a mapping from a value domain to a value range. It can be viewed as a set of ordered pairs which relate each element in the domain uniquely with a corresponding element in the range. Imperative programming language functions, on the other hand, are described as algorithms which specify how to compute the range value from a domain value with a pre-scribed series of steps.

One final characteristic of imperative languages is that repetition— loops— are used extensively to compute desired values. Loops are used to scan through a sequence of memory locations such as arrays, or to accumulate a value in a given variable. In contrast, in mathematical functions, values are computed using function application. Recursion is used in place of iteration. Function composition is used to build more powerful functions.

Mathematical and programming functions

A function is a rule for mapping (or associating) members of one set (the do-main set) to those of another (the range set). For example, the function “ square” might map elements of the set of integer numbers to the set of inte-ger numbers. A function definition specifies the domain, the range, and the mapping rule for the function. For example, the function definition

square(x) ≡ x*x, x is an integer number

defines the function named “ square” as the mapping from integer numbers to integer numbers. We use the symbol “ ≡” for “ is equivalent to.” In this defini-tion, *x* is a *parameter*. It stands for *any* member of the domain set.

Once a function has been defined, it can be *applied* to a particular element of the domain set: the application yields (or *results* in, or *returns*) the associated element in the range set. At application time, a particular element of the domain set is specified. This element, called the *argument*, replaces the parameter in the definition. The replacement is purely textual. If the defini-tion contains any applications, they are applied in the same way until we are left with an expression that can be evaluated to yield the result of the original application. The application

square (2)

results in the value 4 according to the definition of the function square.

Many mathematical functions are defined recursively, that is, the definition of the function contains an application of the function itself. For example, the standard mathematical definition of factorial is:

n! ≡ **if** n = 0 **then** 1 **else** n * (n - 1)!

As another example, we may formulate a (recursive) function to determine if a number is a prime:

prime (n) ≡ **if** n = 2 **then** true **else** p (n, n **div** 2)

where function p is defined as:

```
p (n, i) ≡ if (n mod i) = 0 then false
           else if i = 2 then true
                else p (n, i - 1)
```

Notice how the recursive call to p(n, i-1) takes the place of the next iteration of a loop in an imperative program. Recursion is a powerful problem-solving technique. It is a used heavily when programming with functions.

Principles of functional programming

A functional programming language has three primary components:

1. A set of data objects. Traditionally, functional programming languages have provided a single high level data structuring mechanisms such as a list or an array.
2. A set of built-in functions. Typically, there are a number of functions for manipulating the basic data objects. For example, LISP and ML provide a number of functions for building and accessing lists.
3. A set of functional forms (also called high-order functions) for building new functions. A common example is function composition. Another common example is function reduction. *Reduce* applies a binary function across successive elements of a sequence.

The execution of functional programs is based on two fundamental mechanisms: binding and application. *Binding* is used to associate values with names. Both data and functions may be used as values. Function application is used to compute new values.

In this section we will first review these basic elements of functional pro-grams using the syntax of ML. We will then introduce Lambda calculus, a simple calculus that can be used to model the behavior of functions by defining the semantics of binding and application precisely.

Values, bindings, and functions

As we said, functional programs deal primarily with values, rather than vari-ables. Indeed, variables denote values. For example 3, and "a " are two con-stant values. A and B are two variables that may be *bound* to some values. In ML we may bind values to variables using the binding operator =. For example

```
val A = 3;

val B = "a";
```

The ML system maintains an environment that contains all the bindings that the program creates. A new binding for a variable may hide a previous bind-ing but does not replace it. Function calls also create new bindings because the value of the actual parameter is bound to the name of the formal parameter.

Values need not be just simple data values as in traditional languages. We may also define values that are functions and bind such values to names:

```
val sq = fn(x:int) => x*x;
sq 3;
```

will first bind the variable sq to a function value and then apply it to 3 and print 9. We may define functions also in the more traditional way:

```
fun square (n:int) = n * n;
```

We may also keep a function anonymous and just apply it without binding it to a name:

```
fn(x:int) = x*x) 2;
```

We may of course use functions in expressions:

```
2 * sq (A);
```

will print the value of the expression $2A^2$.

The role of iteration in imperative languages is played by recursion in functional languages.

```
int fact(int n)
{   int i=1;
    assert (n>0);
     {for (int j=n;  j>1; ++j)
         i= i*n;
    return i;
fun fact(n) =
    if n = 0 then 1
    else n*fact(n-1);
}
```

FIGURE . Definition of factorial in C++ and ML

For example, Figure shows the function factorial written in C++ using iteration and ML using recursion.

We saw in Chapter 4 that functions in ML may also be written using patterns and case analysis. The factorial program in the figure may be written as com-posed of two cases, when the argument is 0 and when it is not:

```
fun fact(n) =
        fact(0) = 1
    |   n*fact(n-1);
```

In addition to function definition, functional languages provide functional forms to build new functions from existing functions.

Lambda calculus: a model of computation by functions

In the previous section, we have seen the essential elements of programming with functions: binding, function definition, and function application. As opposed to an imperative language in which the semantics of a program may be understood by following the sequence of steps specified by the program,

Lambda *expressions* represent values in the lambda calculus. There are only three kinds of expressions:

1. An expression may be a single identifier such as x.
2. An expression may be a function definition. Such and expression has the form λ x.e which stands for the expression e with x designated as a *bound* variable. The expression e represents the body of the function and x the paramter. The expression e may contain any of the three forms of lambda expressions. Our familiar square function may be written as λ x.x*x.
3. An expression may be a function application. A function application has the form e1 e2 which stands for expression e1 applied to expression e2. For example, our square function may be applied to the value 2 in this way: (λ x.x*x) 2. Informally, the result of an application can be derived by replacing the parameter and evaluating the resulting expression.

(λ x.x*x) 2=

2*2 =

4

In a function definition, the parameters following the " λ " and before the "." are called *bound* variables. When the lambda expression is applied, the occur-rences of these variables in the expression following the “ .” a re replaced by the arguments. Variables in the definition that are not bound are called *free* variables. Bound variables are like local variables, and free variables are like nonlocal variables that will be bound at an outer level.

Lambda calculus capture the behavior of functions with a set of rules for rewriting lambda expressions. The rewriting of an expression models a step in the computation of a function. To apply a function to an argument, we rewrite the function definition, replacing occurrences of the bound variable by the argument to which the function is being applied.

Representative functional languages

In this section, we examine pure LISP, APL, and ML. LISP was the first functional programming language. The LISP family of languages is large and popular. LISP is a highly dynamic language, adopting dynamic scoping and dynamic typing, and promoting the use of dynamic data structures. Indeed garbage collection was invented to deal with LISP’s heavy demands on dynamic memory allocation. One of the most popular descendants of LISP is Scheme, which adopts static scope rules.

APL in an expression oriented language. Because of the value-orientation of expressions, it has many functional features. As opposed to LISP’s lists, the APL data structuring mechanism is the multidimensional array.

ML is one of the recent members of the family of functional programming languages that attempt to introduce a strong type system into functional programming. We will examine ML in more detail because of its interesting type structure. In the next section, we look at C++ to see how the facilities of a conventional programming language may be used to implement functional programming techniques.

Most functional programming languages are *interactive*: they are supported by an interactive programming system. The system supports the immediate execution of user commands. This is in line with the value-orientation of these languages. That is, the user types in a command and the system immediately responds with the resulting value of the command.

ML

.   .

ML starts with a functional programming foundation but adds a number of features found to be useful in the more conventional languages. In particular, it adopts polymorphism to support the writing of generic components;

Bindings, values, and types

We have seen that establishing a binding between a name and a value is an essential concept in functional programming. Every value in ML has an associ-ated type. For example, the value 3 has type int and the value fn(x:int) =>x*x has type int->int which is the signature of the functional value being defined.

We may also establish new scoping levels and establish local bindings within these scoping levels. These bindings are established using let expressions:

```
let x = 5
in 2*x*x;
```

evaluates to 50. The name x is bound only in the expression in the let expression. There is another similar construct for defining bindings local to a series of other declarations:

```
local
    x = 5
in
   val sq  = x*x
   val cube = x*x*x
end;
```

Such constructs may be nested, allowing nested scoping levels. ML is stati-cally scoped. Therefore, each occurrence of a name may be bound statically to its declaration.

Functions in ML

In ML, we can define a function without giving it a name just as a lambda expression. For example, as we have seen:

```
fn(x, y):int => x*y
```

is a value that is a function that multiplies its two arguments. It is the same as the lambda expression λ x,y.x*y. We may pass this value to another function as argument, or assign it to a name:

```
val intmultiply = fn(x, y):int => x*y;
```

The type of this function is fn:int*int->int.

We have seen that functions are often defined by considering the cases of the input arguments. For example, we can find the length of a list by considering the case when the list is empty and when it is not:

```
fun length(nil) = 0
   |   length([_::x]) = 1+length(x);
```

The two cases are separated by a vertical bar. In the second case, the under-score indicates that we do not care about the value of the head of the list. The only important thing is that there exists a head, whose value we will discard.

List structure and operations

The *list* is the major data structuring mechanism of ML; it is used to build a finite sequence of values of the same type. Square brackets are used to build lists: [2, 3, 4], ["a", "b", "c"], [true, false]. The empty list is shown as [] or nil. A list has a recursive structure: it is either nil or it consists of an element followed by another list. The first element of a nonempty list is known as its *head* and the rest is known as its *tail*.

There are many built-in list operators. The two operators hd and tl return the head and tail of a list, respectively. So: hd([1,2,3]) is 1 and tl([1,2,3]) is [2,3]. Of course, hd and tl are polymorphic. The construction operator :: takes a value and a list of the same type of values and returns a new list: returns [1,2,3]. We can combine two lists by concatenation: [1,2]@[3] is [1,2,3].

```
fun reverse(L) = reverse([]) = []
   |   reverse(x::xs) = reverse(xs) @ [x]
```

Let us write a function to sort a list of integers using insertion sort:

```
fun sort(L) = sort([]) = []
    |   sort(x::xs) = insert (x,xs)
fun insert(x, L) = insert (x,[]) = [x]
    |   insert (x:int, y::ys) =
            if x < y then x::y::ys
                    else y::insert(x,ys);
```

The recursive structure of lists makes them suitable for manipulation by recursive functions. For this reason, functional languages usually use lists or other recursive structures as a basic data structuring mechanism in the lan-guage.

Type system

Unlike LISP and APL, ML adopts a strong type system. Indeed, it has an innovative and interesting type system. It starts with a conventional set of built-in primitive types: bool, int, real, and string. Strings are finite sequences of characters. There is a special type called unit which has a single value denoted as (). It can be used to indicate the type of a function that takes no arguments.

There are several built-in type constructors: lists, records, tuples, and func-tions. A list is used to build a finite sequence of values of a single type. The type of a list of T values is written as T list. For example, [1,2,3] is an int list and ["a","b","cdef"] is a string list. An empty list is written as nil or []. The type of an empty list is ’t list. ’t is a *type variable* which stands for any type. The empty list is a polymorphic object because it is not specifically an empty or an empty bool list. The expression ’t list is an example of a polymorphic type expression (called *polytype* in ML).

Tuples are used to build Cartesian products of values of different types. For example, (5, 6) is of type int*int and (true, "fact", 67) is of type bool*string*int. We can of course use lists as elements of tuples: (true, []) is of type bool* (t’ list).

Records are similar to Pascal records: they are constructed from named fields. For example, {name="Darius", id=56789} is of type {name: string, id: int}. Tuples are special cases of records in which fields are labeled by integers starting with 1. The equality operation is defined for records based on comparing cor-responding fields, that is, the fields with the same names.

In addition to the built-in type constructors, the programmer may define new type constructors, that is, define new types. There are three ways to do this: type abbreviation, datatype definition, and abstract data type definition. The simplest way to define a new type is to bind a type name to a type expression. This is simply an abbreviation mechanism to be able to use a name rather than the type expression. Some examples are:

```
type intpair = int * int;

type ’a pair = ’a * ’a;

type boolpair = bool pair;
```

In the second line, we have defined a new polymorphic type called pair which is based on a type ’a. The type pair forms a Cartesian product of two values of type ’a. We have used this type in the third line to define a new monomorphic type.

Modules

We have already seen an example of an ML module in ML modules are separately compilable units. There are three major building blocks:

1. A *structure* is the encapsulation unit. A structure contains a collection of definitions of types, datatypes, functions, and exceptions.
2. A *signature* is a type for a structure. A signature is a collection of type information about some of the elements of a structure: those we wish to export. We may associate more than one signature with a structure.
3. A functor is an operation that combines one or more structures to form a new structure.

As an example of a module, Figure 86 shows the stack definition of encapsulated in a structure. The figure defines a polymorphic stack implemented as a list.

```
structure Stack = struct

    exception Empty;


    val create = [];

    fun push(x, stack xs) = stack (x::xs);
    fun pop  (stack nil) = raise Empty;
    |    pop (stack [e]) = []
    |    pop (stack [x::xs]) = stack [xs];
    fun top(stack nil) = raise Empty;
    |    top(stack [x::xs]) = x;
    fun lenght(stack []) = 0
    |    length (stack [x::xs]) = length (stack [xs]) + 1;
end;
```

Figure: A stack module in ML

LISP

The original LISP introduced by John McCarthy in 1960, known as pure LISP, is a completely functional language. It introduced many new program-ming language concepts, including the uniform treatment of programs as data, conditional expressions, garbage collection, and interactive program execution. LISP used both dynamic typing and dynamic scoping. Later ver-sions of LISP, including Scheme, have decided in favor of static scoping. Common Lisp is an attempt to merge the many different dialects of LISP into a single language. In this section, we take a brief look at the LISP family of languages.

Data objects

LISP was invented for artificial intelligence applications. It is referred to as a language for symbolic processing. It deals with symbols. Values are repre-sented by symbolic expressions (called *S-expressions*). An expression is either an *atom* or a *list*. An atom is a string of characters (letters, digits, and others). The following are atoms:

```
A
AUSTRIA
68000
```

A list is a sequence of atoms or lists, separated by space and bracketed by parentheses. The following are lists:

(FOOD VEGETABLES DRINKS)

((MEAT CHICKEN) (BROCCOLI POTATOES TOMATOES) WATER)

(UNC TRW SYNAPSE RIDGE HP TUV)

The empty list " ()" , also called NIL. The truth value false is represented as () and true as T. The list is the only mechanism for structuring and encoding information in pure LISP. Other dialects have introduced most standard data structuring mechanisms such as arrays and records.

Functions

There are very few primitive functions provided in pure LISP. Existing LISP systems provide many functions in libraries. It is not unusual Such libraries may contain as many as 1000 functions.

QUOTE is the identity function. It returns its (single) argument as its value. This function is needed because a name represents a value stored in a loca-tion. To refer to the value, we use the name itself; to refer to the name, we use the identity function. Many versions of LISP use 'A instead of the verbose QUOTE A. We will follow this scheme.

The QUOTE function allows its argument to be treated as a constant. Thus, 'A in LISP is analogous to "A" in conventional languages.

Examples

(QUOTE A) = 'A = A

(QUOTE (A B C)) = '(A B C) = (A B C)

ATOM tests its argument to see if it is an atom. NULL, as in ML, returns true if its argument is NIL. EQ compares its two arguments, which must be atoms, for equality.

Examples:

```
(ATOM ('A)) = T
(ATOM ('(A))) = NIL
(EQ ('A) ('A)) = T
(EQ ('A) ('B)) = NIL
```

The function COND serves the purpose of if-then-else expressions. It takes as arguments a number of (predicate, expression) pairs. The expression in the first pair (in left to right order) whose predicate is true is the value of COND.

Example:

```
(COND ((ATOM '(A))) 'B) (T 'A) = A
```

The first condition is false because (A) is not an atom. The second condition is identically true. The COND function, known as the McCarthy conditional, is the major building block for user-defined functions.

Function definition is based on lambda expressions. The function

$\lambda x,y.x+y$

is written in LISP as

```
(LAMBDA (X Y) (PLUS X Y))
```

Function application also follows lambda expressions.

```
((LAMBDA (X Y) (PLUS X Y)) 2 3)
```

binds X and Y to 2 and 3, respectively, and applies PLUS yielding 5.

Functional forms

Function composition was the only technique for combining functions pro-vided by original LISP. For example, the “ to_the_fourth” function can be defined in LISP as

```
(LAMBDA(X) (SQUARE (SQUARE X)))
```

(We assume SQUARE has been defined.) All current LISP systems, however, offer a functional form, called MAPCAR, which supports the application of a function to every element of a list. For example

(MAPCAR TOTHEFOURTH L) raises every element of the list L to the fourth power.

One of the most remarkable points about LISP is the simplicity and elegance of its semantics. In less than one page, McCarthy was able to describe the entire semantics of LISP by giving an interpreter for LISP written in LISP itself. The interpreter is called eval.

APL

APL was designed by Kenneth Iverson at Harvard University during the late 1950s and early 1960s. Even though APL relies heavily on the assignment operation, its expressions are highly applicative. We will only look at these features here to see the use of functional features in a statement-oriented lan-guage.

Objects

The objects supported by APL are scalars, which can be numeric or character, and arrays of any dimension. An array is written as sequence of space-sepa-rated elements of the array. Numeric 0 and 1 may be interpreted as boolean values. APL provides a rich set of functions and a few higher-order functions for defining new functions.

The assignment operation (←) is used to bind values to variables. On assign-ment, the variable takes on the type of the value being assigned to it. For example, in the following, the variable X takes on an integer, a character, and an array, in successive statements:

```
X ← 123;
X ← 'b';

X ← 5 6 7 8 9;
```

The assignment is an operation that produces a value. Therefore, as in C, it may be used in expressions:

X ← (Y ← 5 6 7 8 9) × (Z ← 9 9 7 6 5);

W← Y - Z;   will set the value of Y to 5 6 7 8 9, Z to 9 9 7 6 5, and W to -4 -3 0 2 4.

Functions

In contrast to pure LISP, APL provides a large number of primitive functions (called *operations* in APL terminology). An operation is either monadic (tak-ing one parameter) or dyadic (taking two parameters).

All operations that are applicable to scalars also distribute over arrays. Thus, A x B results in multiplying A and B. If A and B are both scalars, then the result is a scalar. If they are both arrays and of the same size, it is element-by-ele-ment multiplication. If one is a scalar and the other an array, the result is the multiplication of every element of the array by the scalar. Anything else is undefined.

The usual arithmetic operations, +, -, ´, ¸, | (residue), and the usual boolean and relational operation,€ Ù, Ú, ~, <, £, =, >, ³, ¹, are provided. APL uses a number of arithmetic symbols and requires a special keyboard.

There are a number of useful operations for manipulating arrays. The opera-tion “ i” is a “ generator” (or constructor, using ML terminology) and can be used to produce a vector of integers. For example, ι5 produces

1 2 3 4 5

The operation “ ;” concatenates two arrays. So i4; i5 results in

1 2 3 4 1 2 3 4 5

The operation “ r” uses its left operands as dimensions to form an array from the data given as its right operands. For example:

$$2\ 2\ \rho\ 1\ 2\ 3\ 4 \equiv \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$$

and

An APL Program

As an example of the power of functional programming, in this section we will look at how we may derive an APL program to compute prime numbers in the range 1 to N. Despite the limited exposure to APL provided in this sec-tion, we have seen enough to construct the desired program.

The style of programming emphasizes exploiting arrays and expressions rather than scalars, assignments, and iteration. Because of the array orienta-tion of APL, we plan to produce a vector of prime numbers. We can start with a vector of numbers in the range 1 to N and compress it, using the compress operator, to remove the nonprimes. In other words, our task is to find the vec-tor of boolean expressions in the following APL program:

```
vector of boolean expressions /  ιN
```

We can start with the definition of a prime number: a number that is divisible only by 1 and itself. So, for each number in the range of interest, 1 to N, we can (a) divide it by all the numbers in the range and (b) select those which are divisible only by two numbers.

Step (a) can be done with the residue operation and an outer product:

```
(ιN)°.|(ιN)
```

The result of this operation will be a vector of remainders. We are interested in whether the remainder is equal to 0:

```
0 = (ιN)°.|(ιN)
```

Now we have a boolean two-dimensional matrix indicating whether the num-bers were divisible (1) or not (0).

In step (b), we want to see how many times the number was divisible, that is,
the number of 1's in each row:

```
+/[2] 0 = (ιN) °.| (ιN)
```

But we are only interested in those rows that have exactly two 1's.

```
2 = (+/[2] 0 = (ιN) °.| (ιN))
```

<u>Functional programming in C++</u>

It is interesting to ask to what degree traditional programming languages can support functional programming techniques. It turns out that the combination of classes, operator overloading, and templates in C++ provides a surprisingly powerful and flexible support for programming with functions. In this sec-tion, we explore these issues.

Functions as objects

A C++ class encapsulates an object with a set of operations. We may even overload existing operators to support the newly defined object. One of the operator we can overload is the application operator, i.e. parentheses. This can be done by a function definition of the form: operator()(parameters...){...}. We can use this facility to define an object that may be applied, that is, an object that behaves like a function. The class Translate whose outline is shown in Figure is such an object. We call such objects function or functional object. They are defined as objects but they behave as functions.

```
...definitions of types word and dictionary

class Translate {

private: ...;

public:

   word operator()(dictionary& dict, word w)

   {

      //  look up word w in dictionary dict
      //  and return result
   }

}
```

Outline of a function object in C++

We may declare and use the object Translate in this way:

```
Translate Translator(); //construct a Translate object
```

```
cout << Translate(EnglishGermanDict, " university" );
```

which would presumably print " universitaet" , if the dictionary is correct.

The ability to define such objects means that we have already achieved the major element of functional programming: we can construct values of type function in such a way that we can assign them to variables, pass them as

Functional forms

Another major element of functional programming is the ability to define functions by composing other functions. The use of such high-order functions is severely limited in conventional languages and they are indeed one of the distinguishing characteristics of functional languages.

```
...definitions of types word and dictionary

class Translate {

private:

    dictionary D; //local dictionary

public:

    Translate(dictionary& d)

        {D = d;}

    word operator()(word w)

    {

        //  look up word w in dictionary D
        //  and return result
    }

}

...

//construct a German to English translator


Translate GermanToEnglish (GermanEnglishDictionary);
//construct a German to English translator

Translate EnglishToItalian (EnglishItalianDictionary);

...

cout << EnglishToItalian (GermanToEnglish(" universitaet" ));

...
```

Outline of a partially instantiated function object in C++

The 1995 ANSI proposal for the C++ standard library contains a number of function objects and associated templates to support a functional style of programming. For example, it includes a predicate function object greater which takes two arguments and returns a boolean value indicating whether the first argument is greater than the second. We can use such function objects, for example, as a parameter to a sort routine to control the sorting order. We con-struct the function object in this way: greater<int>() or less<int>().

The library also includes a higher-order function find_if, which searches a sequence for the first element that satisfies a given predicate. This find_if takes three arguments, the first two indicate the beginning and end of the sequence and the third is the predicate to be used. Find_if uses the iterators that we discussed in Chapter 5. Therefore, it is generic and can search arrays, lists, and any other linear sequence that provides a pointer-like iterator object. Here, we will use arrays for simplicity. To search the first 10 elements of array a for an element that is greater than 0, we may use something like the following statement:

```
int* p= find_if (a, a+10, "...positive..." ); //not right******
```

What function can we use to check for positiveness? We need to check that something is greater than 0. Given template function objects such as greater, we can build new functions by binding some of their parameters. The library provides *binder* templates for this purpose. There is a binder for binding the first argument of a template function object and a binder for binding the sec-ond argument. For example, we might build a predicate function positive from the function object greater by binding its second argument to 0 in the following way:

```
bind2nd<int>(greater<int>, 0)
```

The library also provides the usual high-order functions such as reduce, accu-mulate, and so on for sequences. The combination of the high level of genericity for sequences and the template function objects to a great degree enable the adoption of a functional style of programming in C++.

Type inference

The template facility of C++ provides a surprising amount of type inference. For example, consider the polymorphic max function given in Figure 90. First, the type of the arguments is simply stated to be of some class T. The

```
template <class T>
T max (T x, T y)
    {if (x>y) return x;
        else return y;
}
```

A C++ generic max function

C++ compiler accepts such a definition as a polymorphic function parameter-ized by type T. We have seen that the ML type inferencing scheme rejects such a function because it cannot infer the type of the operator > used in the function definition. It forces the programmer to state whether T is int or float. C++, on the other hand, postpones the type inferencing to template instantia-tion time. Only when max is applied, for example in an expression ...max(a, b), does C++ do the required type checking. This scheme allows C++ to accept such highly generic functions and still do static type checking. At function definition time, C++ notes the fact that the function is parametric based on type T which requires an operation > and assignment (to be able to be passed and returned as arguments). At instantiation time, it checks that the actual parameters satisfy the type requirements.

Logic and rule-based languages

The "what" versus "how" dilemma: specification versus implementation

A software development process can be viewed abstractly as a sequence of phases through which system descriptions progressively become more and more detailed. Starting from a software requirements specification, which emphasizes *what* the system is supposed to do, the description is progressively refined into a procedural and executable description, which describes *how* the problem actually is solved mechanically. Intermediate steps are often standardized within software development organizations, and suitable notations are used to describe their outcomes (software artifacts). Typically, a design phase is specified to occur after requirements specification and before implementation, and suitable software design notations are provided to document the resulting software architecture. Thus the "what" stated in the requirements is transformed into the "how" stated in the design document, i.e., the design specification can be viewed as an abstract implementation of the requirements specification. In turn, this can be viewed as the specification for the subsequent implementation step, which takes the design specification and turns it into a running program.

In their evolution, programming languages have become increasingly higher level. For example, a language like Ada, Eiffel, and C++ can be used in the design stage as a design specification language to describe the modular structure of the software and module interfaces in a precise and unambiguous way, even though the internals of the module (i.e., private data structures and algorithms) are yet to be defined. Such languages, in fact, allow the module specification (its interface) to be given and even compiled separately from the module implementation. The specification describes "what" the module does by describing the resources that it makes visible externally to other modules; the implementation describes "how" the internally declared data strucures and algorithms accomplish the specified tasks.

All of the stated steps of the process that lead from the initial requirements specification down to an implementation can be guided by suitable systematic methods. They cannot be done automatically, however: they require engi-neering skills and creativity by the programmer, whose responsibility is to map– translate– requirements into executable (usually, procedural) descriptions. This mapping process is time-consuming, expensive, and error-prone activities.

An obvious attempt to solve the above problem is to investigate the possibility of making specifications directly executable, thus avoiding the translation step from the specification into the implementation. Logic programming tries to do exactly that. In its simplest (and ideal) terms, we can describe logic programming in the following way: A programmer simply declares the properties that describe the problem to be solved. The problem description is used by the system to solve the problem (*infer a solution*). To denote its distinctive capabilities, the run-time machine that can execute a logic language is often called an *inference engine*.

In logic programming, problem descriptions are given in a logical formalism, based on first-order predicate calculus. The theories that can be used todescribe and analyze logic languages formally are thus naturally rooted into mathematical logic. Our presentation, however, will avoid delving into deep mathematical concepts, and will mostly remain at the same level in which more conventional languages were studied.

The above informal introduction and motivations point out why logic pro-gramming is often said to support a *declarative* programming paradigm. As we will show, however, existing logic languages, such as PROLOG, match this description only partially. To make the efficiency of the program execution acceptable, a number of compromises are made which dilute the purity of the declarative approach. Efficiency issues affect the way programs are writ-ten; that is, the programmer is concerned with more than just the specification of what the program is supposed to do. In addition, non-declarative language features are also provided, which may be viewed as directions provided by the programmer to the inference engine. These features in general reduce the clarity of program descriptions.

A first example

In order to distinguish between specification and implementation, and to introduce logic programming, let us specify the effect of searching for an ele-ment x in a list L of elements. We introduce a predicate is_in (x, L) which is true whenever x is in the list L. The predicate is described using a self-explaining hypothetical logic language, where operator "• " denotes the concatenation of two lists and operator [ ] transforms an element into a list containing it and "iff" is the conventional abbreviation for "if and only if.".

```
for all elements x and lists L: is_in (x, L)  iff
    L = [x]
or
    L = L1 • L2 and
        (is_in (x, L1) or is_in (x, L2))
```

The above specification describes a binary search in a declarative fashion. The element is in the list if the list consists exactly of that element. Otherwise, we can consider the list as decomposed into a left sublist and a right sublist, whose concatenation yields the original list. The element is in the list, if it is in either sublist. Let us now proceed to an implementation of the above specification. Besides other details, an implementation of the above specification must decide

- how to split a list into a right and a left sublist. An obvious choice is to split it into two sublists of either the same length, or such that they differ by at most one;
- how to store the elements in the list. An obvious choice is to keep the list sorted, so that one can decide whether to search the left or the right sublist and avoid searching both;
- how to speed up the search. Instead of waiting until a singleton list is obtained via repeated splitting, the algorithm can check the element that separates the two sublists. If the separator equals the desired element, the search can stop. Otherwise, it proceeds to check either in the right or in the left sublist generated by the splitting, depending on the value of the separator.

A possible C++ implementation of the specification is shown in Figure.By looking carefully at both the logic specification and the C++ implementation, one can appreciate the differences between the two in terms of ease of writ-ing, understandability, and self-confidence in the correctness of the description with respect to the initial problem.

Instead of transforming the specification into an implementation, one might wonder whether the specification can be directly executed, or used as a start-ing point for a straightforward derivation process yielding an implementation. To do so, we can read the above declarative specification procedurally as fol-lows:

> Given an element x and a list L, in order to prove that x is in L, proceed as follows
>
> (1) prove that L is [x];
>
> (2) otherwise split L into L1 • L2 and prove one of the following: (2.1) x is in L1, or
> (2.2) x is in L2

A blind mechanical executor which follows the procedure can be quite ineffi-cient, especially if compared to the C++ program. This is not surprising. Direct execution is less efficient than execution of an implementation in a tra-ditional procedural language, but this is the obvious price we pay for the sav-ings in programming effort.

```
int binary_search (const int val, size, const int array[ ]) {

   //return the index of the desired value val, if it is there
   //otherwise return -1
   if size ð  0 {

      return (-1);

   }

   int high = size; // the portion of array to search is

   int low = 0;     // low. .high-1

   for ( ; ; ) {

      int mid = (high + low) / 2;

      if (mid = low) {

         // search is finished

         return (test != array [low]) ? -1 : mid;

      }

      if (test < array [mid]) {

         high = mid;

      }

      else if (test > array [mid]) {

         low = mid;

      else {
```

```
                return mid;
            }
        }
    }
```

FIGURE A C++ implementation of binary search

Principles of logic programming

To understand exactly how logic programs can be formulated and how they can be executed, we need to define a possible reference syntax, and then base on it a precise specification of semantics.

Preliminaries: facts, rules, queries, and deductions

Although there are many syntactic ways of using logic for problem descriptions, the field of logic programming has converged on PROLOG, which is based on a simple subset of the language of first-order logic. Hereafter we will gradually introduce the notation used by PROLOG.

The basic syntactic constituent of a PROLOG program is a *term*. A term is a constant, a variable, or a compound term. A compound term is written as a *functor symbol* followed by one or more arguments, which are themselves terms. A *ground term* is a term that does not contain variables. Constants are written as lower-case letter strings, representing atomic objects, or strings of digits (representing numbers). Variables are written as strings starting with an upper-case letter. Functor symbols are written as lower-case letter strings.

```
alpha

125

X

abs (-10, 10)

abs (suc (X), 5)
```

The constant [ ] stands for the empty list. Functor "." constructs a list out of an element and a list; the element becomes the head of the constructed list. For example, .(alpha, [ ]) is a list containing only one atomic object, alpha. An equivalent syntactic variation, [alpha, [ ]], is also provided. Another example would be

```
.(15, .( toot, .(duck, donald)))
```

which can also be represented as

```
[15, [toot, [duck, donald]]]
```

The notation is further simplified, by allowing the above list to be written as

```
[15, toot, duck, donald]
```

and also as

```
[15 | [toot, duck, donald]]
```

In general, the notation

```
[X | Y]
```

stands for the list whose head element is X and whose tail list is Y.

A predicate is represented by a compound term. For example

```
less_than (5, 99)
```

states the "less than" relationship between objects 5 and 99.

PROLOG programs are written as a sequence of *clauses*. A clause is expressed as either a single predicate, called *fact*, or as a *rule* (called *Horn clause*) of the form

```
conclusion :- condition
```

where :- stands for "if", conclusion is a single predicate, and condition is a con-junction of predicates, that is, a sequence of predicates separated by a comma, which stands for the logical and. Facts can be viewed as rules without a condi-tion part (i.e., the condition is always true). Thus the term "rule" will be used to indicate both facts and rules, unless a distinction will be explicitly made. A rule’s conclusion is also called the rule’s *head*. Clauses are implicitly quanti-fied universally. A PROLOG rule

```
conclusion :- condition
```

containing variable X1, X2, . . ., Xn would be represented in the standard notation of mathematical logic as

"X1, X2, . . ., Xn (condition É conclusion)

where É is the logical implication operator. In a procedural program, it would be represented as

if condition **then** conclusion;

For example, the following program

```
length ([ ], 0).   --this is a fact
length ([X | Y], N) :- length (Y, M), N = M + 1. --this is a rule
```

says that

- the length of the null string is zero,
- for all X, Y, N, M, if M is the length of list Y and N is M + 1, then the length of a nonnull string with head X and tail Y is one more than the length of Y.

An abstract interpretation algorithm

As we mentioned earlier, the abstract processor must be able to take a query as a goal to be proven, and match it against facts and rule heads. The matching process, which generalizes the concept of procedure call, is a rather elaborate operation, called *unification*, which combines pattern matching and binding of variables.

Unification applies to a pair of terms t1 (representing the goal to prove) and t2 (representing the fact or rule’s head with which a match is tried). To define it, we need a few other background definitions. Term t1 is said to be *more gen-eral* than t2 if there is a substitution µ such that t2 = µ (t1), but there is no substi-tution ¼ such that t1 = ¼ (t2). Otherwise, they are said to be *variants*, i.e., they are equal up to a renaming of variables.

Two terms are said to unify if a substitution can be found that makes the two terms equal. Such a substitution is called a *unifier*. For example, the substitu-tion

s1 = {<X, a>, <Y, b>, <Z, b>}

is a unifier for the terms f (X, Y) and f (a, Z). A *most general unifier* is a unifier

µ such that µ (t1) = µ (t2) is the most general instance of both t1 and t2. It is easy to prove that all most general unifiers are variants. We will therefore speak of "the" most general unifier (MGU), assuming that it is unique up to a renaming of its variables. In the previous example, s1 is not the MGU. In fact the substi-tution

s2 = {<X, a>, <Y, W>, <Z, W>}

is more general than s1, and it is easy to realize that no unifier can be found that is more general than s2.

MGUs are computed by the unification algorithm shown in Figure. The algorithm keeps the set of pairs of terms to unify in a working set which is initialized to contain the pair <t1, t2>.The algorithm is written in a self-explain-ing notation. If the two terms given to the algorithm do not unify, the exception unification_fail is raised.

The algorithm operates on two terms t1 and t2 and returns their MGU.

It raises an exception if the unification fails.

```
MGU = { }; --MGU is the empty set
WG = {<t1, t2>}; --working set initialized
repeat
    extract a pair <x1, x2> from WG;
    case
        • x1 and x2 are two identical constants or
            variables: do nothing
        • x1 is a variable that does not occur in x2:
            substitute x2 for x1 in any pair of WG and in MGU;
        • x2 is a variable that does not occur in x1:
            substitute x1 for x2 in any pair of WG and in MGU;
        • x1 is f (y1, y2, . . ., yn), x2 is f (z1, z2, . . ., zn), and f is a functor and n Š 1:
            insert <y1, z1>, <y2, z2>, . . ., <yn, zn> into WG;
        otherwise
            raise unification_fail;
    end case;
until WG = { } --working set is empty
```

FIGURE. Unification algorithm

when a goal is being solved, it may be necessary to choose one out of several clauses to attempt unification. It may happen that the choice we make eventually results in a failure, while another choice would have led to success. For example, in the case of computation (a) the choice of clause 5 instead of 6 for the first match leads to a failure. Second, when several goals can be selected from SG, there is a choice of which is to be solved first. For example, when the goals to be solved are rel(a, Z1), clos (Z1, f), computations (c) and (d) make their selection in a different order. In general, when we have to choose which of two goals G1 and G2 is to be solved first, it can be shown that if there is a successful computation choosing G1 first, there is also a suc-cessful computation choosing G2 first. The choice can only affect the effi-ciency of searching a solution. From now on, we will assume that whenever a goal unifies with a rule's head, the (sub)goals corresponding to the righthand side of the rule will be solved according to a deterministic policy, from left to right.

PROLOG

PROLOG is the most popular example of a logic language. Its basic syntactic features were introduced informally. As we anticipated, PRO-LOG solves problems by performing a depth-first traversal of the search tree. Whenever a goal is to be solved, the list of clauses that constitutes a program is searched from the top to the bottom. If unification succeeds, the subgoals corresponding to the terms in the righthand side of the selected rule (if any) are solved next, in a predined left-to-right order. This particular way of oper-ating makes the behavior of the interpreter quite sensitive to the way pro-grams are written. In particular, the ordering of clauses and subgoals can influence the way the interpreter works, although from a conceptual view-point clauses are connected by the logical operator "or" and subgoals are con-nected by "and". These connectives do not exhibit the expected commutativity property.

As an example, Figure shows all possible permutations of terms for the closure relation that was discussed. It is easy to verify that any query involving predicate clos would generate a nonterminating computa-tion in case (4). Similarly, a query such as ?-clos (g, c) causes a nonterminating interpretation process in cases (2) and (4), whereas in (1) and (3) the inter-preter would produce NO.

(1)
```
clos (X, Y) :- rel (X, Y).
clos (X, Y) :- rel (X, Z), clos (Z, Y).
```

(2)
```
clos (X, Y) :- rel (X, Y).
clos (X, Y) :- clos (Z, Y), rel (X, Z).
```

(3)
```
clos (X, Y) :- rel (X, Z), clos (Z, Y).
clos (X, Y) :- rel (X, Y).
```

(4)
```
clos (X, Y) :- clos (Z, Y), rel (X, Z).
clos (X, Y) :- rel (X, Y).
```

FIGURE . Variations of a PROLOG program

PROLOG provides several extra-logical features, which cause its departure from a pure logical language. The first fundamental departure is represented by the *cut* primitive, written as "!", which can appear as a predicate in the con-dition part of rules. The effect of the cut is to prune the search space by for-bidding certain backtracking actions from occurring. Its motivation, of course, is to improve efficiency of the search by reducing the search space. It is the programmer's responsibility to ensure that such a reduction does not affect the result of the search. The cut can be viewed as a goal that never fails and cannot be resatisfied. That is, if during backtracking one tries to resatisfy it, the goal that was unified with the lefthand side of the rule fails.

Functional programming versus logic programming

The most striking difference between functional and logic programming is that programs in a pure functional programming language define functions, whereas in pure logic programming they define relations. In a sense, logic programming generalizes the approach taken by relational databases and their languages, like SQL. For example, consider the simple PROLOG program shown in Figure 99, consisting of a sequence of facts. Indeed, a program of this kind can be viewed as defining a relational table; in the example, a mini-database of classical music composers, which lists the composer's name, year of birth, and year of death. (See the sidebar on relational database languages and their relation to logic languages.)

In a function there is a clear distinction between the domain and the range. Executing a program consists of providing a value in the domain, whose cor-responding value in the range is then evaluated. In a relation, there is no pre-defined notion of which is the input domain. In fact, all of these possible queries can be submitted for the program of Figure:

```
?- composer (mozart, 1756, 2001).

?- composer (mozart, X, Y).

?- composer X, Y, 1901).

?- composer (X, Y, Z).
```

In the first case, a complete tuple is provided, and a check is performed that the tuple exists in the database. In the second case, the name of the composer is provided as the input information, and the birth and death years are evalu-ated by the program. In the second case, we only provide the year of death, and ask the program to evaluate the name and year of birth of a composer whose year of death is given as input value. In the fourth case, we ask the sys-tem to provide the name, year of birth, and year of death of a composer listed in the database.

```
composer (monteverdi, 1567, 1643).
composer (bach, 1685, 1750).
composer (vivaldi, 1678, 1741).
composer (mozart, 1756, 1791).
composer (haydn, 1732, 1809).
composer (beethoven, 1770, 1827).
composer (schubert, 1797, 1828).
composer (schumann, 1810, 1856).
composer (brahms, 1833, 1897).
composer (verdi, 1813, 1901).
composer (debussy, 1862, 1918).
```

FIGURE .A PROLOG database

As most functional languages are not purely functional, PROLOG is not a pure logic language. Consequently, it is not fully relational in the above sense. In particular, the choice of the input domains of a query is not always free. This may happen if the program contains relational predicates, assign-ment predicates, or other extralogical features. For example the factorial pro-gram of Figure cannot be invoked as follows ?- fact (X, 6). to find the integer whose factorial is 6. The query would in fact fail, because the extralogical predicate is fails. Similarly, the following query

```
?- max (X,99, 99).
```

for the program fragment of Figure does not yield a value less than or equal to 99, as the logical reading might suggest. It fails, since one of the arguments in the invocation of ð is not bound to a value.

Rule-based languages

Rule-based languages are common tools for developing expert systems. Intu-itively, an *expert system* is a program that behaves like an expert in some restricted application domain. Such a program is usually structured as a *knowledge base (KB)*, which comprises the knowledge that is specific to the application domain, and an *inference engine*. Given the description of the *current situation (CS),* often called the database, expressed as a set of facts, the inference engine tries to match CS against the knowledge base to find the rules that can be *fired* to derive new information to be stored in the database, or to perform some action.

An important class of expert system languages (called *rule-based languages*, or *production systems*) uses the so-called *production rules*. Production rules are syntactically similar to PROLOG rules. Typical forms are:

- if condition then action .

For example, the MYCIN system for medical consultation allows rules of this kind to be written:

```
if
    description of symptom 1, and
    description of symptom 2, and
    . . .
    description of symptom n
then
    there is suggestive evidence (0.7) that the identity of the bacterium is . . .
```

The example shows that one can state the "degree of certainty" of the conclu-sion of a rule. In general, the action part of a production rule can express any action that can be described in the language, such as updating CS or sending messages.

Supposing that knowledge is represented using production rules, it is neces-sary to provide a reasoning procedure (inference engine) that can draw con-clusions from the knowledge base and from a set of facts that represent the current situation. For production rules there are two basic ways of reasoning:

– *forward chaining*, and

– *backward chaining*.

Different rule-based languages provide either one of these methods or both.

In order to understand forward and backward chaining, let us introduce a sim-ple example described via production rules. The knowledge base provides a model of a supervisory system that can be in two different danger states, char-acterized by levels 0 and 1, indicated by the state of several switches and lights:

if s

```
witch_1_on and switch_2_on
    then notify danger_level_0.
if switch_1_on and switch_3_on
    then assert problem_1.
if light_red or alarm_on
    then assert problem_2.
if problem_1 and problem_2
    then notify danger_level_1.
```