Third Edition

Principles Of Programming Languages

Design, Evaluation, and Implementation

Bruce J. MacLennan
University of Tennessee, Knoxville

New York • Oxford
Oxford University Press
1999

Contents

Concept Directory

History, Motivation, and Evaluation

Note: Page numbers denote the entire subsection that begins on the specified page.

Design and Implementation

Principles

Implementation

0 INTRODUCTION

WHAT IS A PROGRAMMING LANGUAGE?

The subject of this book is programming languages, specifically, the principles for their design, evaluation, and implementation. Thus, we must begin by saying what a programming language is.

A programming language is a language intended for the description of programs. Often a program is expressed in a programming language so that it can be executed on a computer. This is not the only use of programming languages, however. They may also be used by people to describe programs to other people. Indeed, since much of a programmer's time is spent reading programs, the understandability of a programming language to people is often more important than its understandability to computers.

There are many languages that people use to control and interact with computers. These can all be referred to as *computer languages.* Many of these languages are used for special purposes, for example, for editing text, conducting transactions with a bank, or generating reports. These special-purpose languages are not *programming* languages because they cannot be used for general programming. We reserve the term *programming language* for a computer language that can be used, at least in principle, to express any computer program.[1] Thus, our final definition is

A programming language is a language that is intended for the expression of computer programs and that is capable of expressing any computer program.

[1] This is not a vague notion. There is a precise theoretical way of determining whether a computer language can be used to express any program, namely, by showing that it is equivalent to a universal Turing machine. This topic is outside the scope of this book.

THE DIFFERENCES AMONG PROGRAMMING LANGUAGES

Since, by definition, any programming language can be used to express any program, it follows that all programming languages are equally powerful—any program that can be written in one can also be written in another. Why, then, are there so many programming languages? And why should one study their differences, when in this very fundamental sense they are all the same? The reason is that, although it's *possible* to write any program in any programming language, it's not equally *easy* to do so. Thus, in this book, we will not be very concerned with the *theoretical* power of programming languages (they're all the same). Rather, we concentrate on their *practical* power, as real tools used by real people. In this regard they will be seen to differ in many important respects. But why devote so much time to just one kind of tool?

IMPORTANCE OF THE STUDY OF PROGRAMMING LANGUAGES

Programming languages are important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science: programming. As a result, the progress of computer science can be traced in the progress of programming languages, and many issues of computer science manifest themselves as programming language issues. This is particularly true in programming methodology, where advances in languages and programming techniques have gone hand in hand. The reason is simple: Programming languages remain the central tool for problem solving in computer science.

INFLUENCE OF LANGUAGES ON PROBLEM SOLVING

The Sapir–Whorf hypothesis is a (still controversial) linguistic theory that states that the structure of language defines the boundaries of thought. Although there is no evidence that the use of a particular language will prevent us from thinking certain thoughts, it is the case that a given language can facilitate or impede certain modes of thought and that languages embody characteristic ways of dealing with the world and other people. When applied to programming languages, the analogous statement is that although no programming language can prevent us from finding certain solutions to a problem, a given language can influence the class of solutions we are likely to see and the frame of mind with which we approach programming, thus subtlely influencing the quality of our programs.

BENEFITS FOR ALL COMPUTER SCIENTISTS

The study of programming languages is important to anyone who uses them, that is, anyone who programs. The reason is that from this study you will learn the motivation for and the use of the most important facilities found in modern programming languages. You will learn the benefits of these facilities, as well as their costs, by studying the techniques used to implement them. This will provide you with a basis for evaluating languages, which will aid you in choosing the best language for your application. The understanding acquired of the motivations for the facilities in a language will enable you to use those facilities to their fullest potential. The repertoire of language mechanisms with which you are familiar will have been increased so that even if the language you must use does not provide the facilities you need, you will be able to simulate them through your knowledge of their implementation.

There are many programming languages now in widespread use—many more than can be taught to you as part of your computer science education. This means that in your computer science career you will be required frequently to learn a new programming language and to put it to effective use. Your speed in learning new languages is one aspect of your versatility as a computer scientist. Fortunately, underneath the surface details most languages are very similar. Therefore, the study of programming languages, by increasing the range of facilities in which you are fluent, will enable you to see more that is familiar in any new language that you encounter. This will speed your learning of new languages.

BENEFITS FOR LANGUAGE DESIGNERS

Although, as indicated above, the study of programming languages is important to all students of computer science, it is especially important to certain disciplines. Obviously, it is important if you are a student of language design. All engineering design is a cumulative process; we learn from the successes and failures of the designs of the past. To this end it is necessary to be familiar with the history of programming languages. As George Santayana said, "Those who cannot remember the past are condemned to repeat it." An understanding of the reasons why certain designs have been tried in the past and later abandoned will help you to develop a sense of good language design and to become skillful in making design trade-offs. As R.W. Hamming has said, "We would know what they thought when they did it." To help you remember the lessons of the past, we have formulated and illustrated a number of *maxims or principles* of good programming language design. The central role these play has dictated the book's title: *Principles of Programming Languages.*

BENEFITS FOR LANGUAGE IMPLEMENTERS

If you are interested in language implementation, you will gain insight into the motivations for various language facilities, thus allowing you to make reasonable implementation trade-offs. Although language implementation is a complicated subject, requiring one or more courses, this book presents the most useful and important techniques for implementing a number of common programming language facilities. These are seminal techniques that can be elaborated to satisfy more stringent requirements or varied to solve related problems; they are a basis for further studies in language implementation.

BENEFITS FOR HARDWARE ARCHITECTS

By understanding the requirements of programming language implementation, hardware architects will gain insight into the ways machines may better support languages. More important, you will learn to design a semantically coherent machine—a machine with complete and coherent sets of data types and operations on those data types. The reason for this is simple. Just as a programming language can be considered a *virtual computer*, that is, a computer implemented in software, so a computer can be considered a programming language implemented in hardware. This view suggests that many of the principles of programming language design can be equally well applied to computer architecture, and indeed they can.

BENEFITS FOR SYSTEM DESIGNERS

Designers of all sorts of software systems (e.g., operating systems and database systems) will learn principles and techniques applicable to all human interfaces. Many software tools including operating system command languages, database systems, editors, text formatters, and debuggers, have many of the characteristics of programming languages, and so the principles you learn here will be applicable to much of your future software design. The study of both language design and implementation is obviously valuable here. Knowledge of programming languages is more directly necessary for designers of file systems, linkage editors, and other software that must interface with programming languages.

BENEFITS FOR SOFTWARE MANAGERS

Finally, if you manage software development efforts, then you will benefit in several ways from the study of programming languages. The project manager often makes decisions regarding the language to be used on a given project, or whether an existing language should be used or extended, or whether a completely new language should be designed. You will be better able to do this if you know what common languages can and cannot do, and if you know the current direction and state of the art of programming language research. You will be better able to make these decisions if you know the costs of designing or extending a language, the costs of implementing a language, and the benefits of various language facilities.

PLAN OF THE BOOK

In 1965 an American Mathematical Association Prospectus estimated that 1700 programming languages were then in use.[2] In the intervening years, many more have been invented.

[2] Quoted in P.J. Landin, "The Next 700 Programming Languages." *Commun. ACM 9*, 3 (March 1966), p. 157.

Clearly, it is impossible to discuss every language, or even a sizable fraction of them. How have we chosen the languages to present in this book?

Certainly, we have chosen languages of actual or potential importance. All other things being equal, we have chosen languages that you are likely to encounter in your career as a computer scientist. But there are other, more important factors in our selection.

An understanding of these factors is implicit in the purpose of this book—which is not to teach you to program in half a dozen programming languages.[3] As noted before, there is little chance that we could teach you just those languages you will later need to know. Rather, our goal is to present the most important principles for the design, evaluation, and implementation of programming languages. To this end, we have chosen languages that will serve as good *case studies* to illustrate these principles.

These principles have developed in a series of historical stages, each being a reaction to the perceived problems and opportunities discovered in the previous stage. For this reason, we have chosen programming languages that are illustrative of the major generations of programming language evolution. Thus, FORTRAN, Algol-60, Pascal, and Ada are representatives of the first, second, third, and fourth programming language generations.[4] We have picked these particular languages because they form a single evolutionary line in the family tree of programming languages.

Since language development is now entering the fifth generation, it is too early to predict what the next stage in programming language evolution will be. Therefore, we have illustrated the fifth generation with representatives of three important new programming paradigms: function-oriented programming (LISP), object-oriented programming (Smalltalk), and logic-oriented programming (PROLOG). It is likely that all three of these paradigms will be important in the years to come.

[3] Thus, there are few exercises in this book that require you to do significant programming in a language under study. Attempting to evaluate a language on the basis of writing a few short programs in it is as misleading as evaluating an automobile by driving it once or twice around the block. Meaningful evaluation requires experience with large programs maintained over long periods, which is impractical in a course such as this. As is often the case, if we are careful, we will learn more by vicarious experience than by direct experience.

[4] Note that some authors use the term *fourth-generation language* to refer to various application generation packages. These are not *programming* languages in the sense referred to previously; we are discussing fourth-generation *programming* languages.

ory in which each subprogram will go. Therefore, the exact addresses of variables and statements are not yet known; we say that they have a later *binding time* because they are bound to addresses at load-time rather than at compile-time. It is for these reasons that the object program is represented in a special *relocatable* format that allows the addresses to be assigned at load-time. Relocatable format is similar to the symbolic statement and variable labels in our pseudo-code.

The second step, *linking*, addresses the need for incorporating *libraries* of already programmed, debugged, and compiled subprograms. Needless to say, the presence of a good library can greatly simplify the programming process, as we saw with floating-point libraries in Chapter 1. The use of libraries means that programs contain *external references* to subprograms in these libraries. Furthermore, these library subprograms may themselves contain external references to other library subprograms. To obtain a complete program, all of these external references must be resolved or *satisfied* by finding the corresponding subprograms in the library. This is the goal of the linking process; its result is usually a file containing all of the parts of the program, still in relocatable format, but with their external references satisfied.

The third step, *loading*, is the process in which the program is placed in computer memory. This requires converting it from relocatable to *absolute* format, that is, it requires binding all code and data references to the addresses of the locations that the code and data will occupy in memory.

The final step, *execution*, is the one in which control of the computer is turned over to the program in memory. Since the program is executed directly rather than interpreted, it has the potential of running much faster.

Compilation Involves Three Phases

The compilation process is obviously one of the most important steps in the processing of a FORTRAN program since it is this step that determines the efficiency of the final program. For a language such as FORTRAN, compilation usually involves three tasks. These may be performed one after the other or interleaved in various ways.

1. **Syntactic analysis:** The compiler must classify the statements and constructs of FORTRAN and extract their parts.
2. **Optimization:** As we have said, efficiency was a prime goal of the original FORTRAN system. For this reason the original FORTRAN system included a sophisticated optimizer whose goal was to produce code as efficient as could be produced by an experienced programmer. Most FORTRAN compilers perform at least a moderate amount of optimization.
3. **Code synthesis:** The final task of compilation is to put together the parts of the object code instructions in relocatable format.

2.3 DESIGN: CONTROL STRUCTURES

Control Structures Govern Primitive Statements

In Section 2.3 we discuss FORTRAN's *control structures*, that is, those constructs in the language that govern the flow of control of the program. We will find that these control structures are elaborations of the control structures found in the pseudo-code of Chapter 1. In the pseudo-code we saw that the purpose of control structures was to direct control to various *primitive* computational and input-output instructions such as ADD and READ. (By a *primitive* operation we mean one that is not expressed in terms of more fundamental ideas in the language.) The situation is similar in FORTRAN; the computational and input-output instructions do the actual data processing work of the program, while the control structures act as "schedulers" or "traffic managers" by directing control to these primitive statements. We will see in later chapters that this organization is common to all *imperative* programming languages.

As we have said (Section 2.1), the ability to write more or less familiar looking algebraic equations was one of the major contributions of FORTRAN. Without doubt, the assignment statement is the most important statement in FORTRAN. In fact, a FORTRAN program can be considered as a collection of assignment statements with provision (i.e., the control structures) for directing control to one or the other of these assignment statements.

Control Structures Were Based on IBM 704 Branch Instructions

FORTRAN was originally designed as a programming language for the IBM 704 computer; it was thought that there would be similar, but different, languages for other computers. Backus has said that he never imagined that FORTRAN would be used on the computers of other manufacturers. It is thus not surprising that the first FORTRAN had many similarities to the 704 instruction set; designers have a tendency to include in a language the features they have previously found useful. This *machine dependence* is a characteristic of first-generation languages.

We can see the machine dependence of FORTRAN's control structures in Figure 2.2, which displays the similarities between FORTRAN II's control structures and the branch instructions of the IBM 704. It is not important that you understand the 704 instructions or even the FORTRAN statements at this point; what is important is the correspondence. It is one reason that FORTRAN has sometimes been called an "assembly language for the IBM

FORTRAN II Statement	704 Branch	
`GOTO n`	`TRA k`	(transfer direct)
`GOTO n, (n1, n2, ..., nm)`	`TRA i`	(transfer indirect)
`GOTO (n1, n2, ..., nm), n`	`TRA i,k`	(transfer indexed)
`IF (a) n1, n2, n3`	`CAS k`	(compare AC with storage)
`IF ACCUMULATOR OVERFLOW n1, n2`	`TOV k`	(transfer on AC overflow)
`IF QUOTIENT OVERFLOW n1, n2`	`TQO k`	(transfer on MQ overflow)
`DO n i = m1, m2, m3`	`TIX d,i,k`	(transfer on index)
`CALL name (args)`	`TSX i,k`	(transfer and set index)
`RETURN`	`TRA i`	(transfer indirect)

Figure 2.2 Similarity of FORTRAN and IBM 704 Branches

704," although some of the more blatantly machine-dependent statements (e.g., `IF QUOTIENT OVERFLOW`) were removed from FORTRAN IV and later versions. This correspondence also explains some of FORTRAN's more unusual control structures, for example, the *arithmetic* `IF`*-statement*

`IF` $(e)\ n_1, n_2, n_3$

evaluates the expression e and then branches to n_1, n_2, or n_3 depending on whether the result of the evaluation is negative, zero, or positive, respectively. This is exactly the function of the 704's `CAS` instruction, which compares the accumulator with a value in storage and then branches to one of three locations. The arithmetic `IF` was not very satisfactory for a number of reasons, including the difficulty of keeping the meaning of the three labels straight and the fact that two of the labels were usually identical (because two-way branches are more common than three-way branches). In later versions of FORTRAN, a more conventional *logical* `IF`*-statement* was added, for example,

```
IF (X .EQ. A(I)) K = I - 1
```

(FORTRAN uses `.EQ.` for the equality relation.)

Machine-dependent features such as the arithmetic `IF` are violations of

> **The Portability Principle**
>
> Avoid features or facilities that are dependent on a particular computer or a small class of computers.

The GOTO Is the Workhorse of Control Flow

Just as in most computers, the transfer (i.e., branch or jump) instruction is the primary means for controlling the flow of execution, so in FORTRAN the `GOTO`-statement and its variants are the fundamental control structures. We will now investigate the implications of this fact on program readability.

In FORTRAN, the `GOTO`-statement is the raw material from which control structures are built. For example, a two-way branch is often implemented with a logical `IF`-statement and a `GOTO`:

```
      IF (condition) GOTO 100
      ... case for condition false ...
      GOTO 200
100   ... case for condition true ...
200   ...
```

We can see that this corresponds to the **if-then-else,** or *conditional* statement of newer programming languages, although the false- and true-branches are placed in the opposite order. If we wish to put them in the same order, we must then negate the condition, which may be confusing:

```
      IF (.NOT. (condition) ) GOTO 100
      ... case for condition true ...
      GOTO 200
100   ... case for condition false ...
200   ...
```

This use of the IF-statement divides the control flow into two cases. For dividing it into more than two cases, a *computed* GOTO is provided, for example,

```
      GOTO (10, 20, 30, 40), I
10    ... handle case 1 ...
      GOTO 100
20    ... handle case 2 ...
      GOTO 100
30    ... handle case 3 ...
      GOTO 100
40    ... handle case 4 ...
100   ...
```

The meaning of this is to branch to statement 10, 20, 30, or 40 if I is 1, 2, 3, or 4, respectively. We can recognize this as the equivalent of the **case**-statement that is included in many contemporary languages.[2]

Both the IF-statement and the computed GOTO are examples of *selection* statements, so called because they select between two or more possible control paths.

Loops can be implemented by various combinations of IF-statements and GOTOs (the DO-loop is discussed later). For example, a *trailing-decision loop* could be written

```
100   ... body of loop ...
      IF (loop not done) GOTO 100
```

and a *leading-decision loop* could be written

```
100   IF (loop done) GOTO 200
      ... body of loop ...
      GOTO 100
200   ...
```

We can recognize these as the **while-do** and **repeat-until** constructs of languages such as Pascal. These constructs are called *indefinite iterations* because the exact number of iterations is not known in advance (i.e., not definite).

One problem with using one statement (GOTO) to build all control structures is that we never know what control structure is intended. For example, in Pascal, when we see **while** we know that a leading-decision loop is intended; when we see **repeat** a trailing-decision loop is intended; and when we see **if** a conditional statement is intended. In FORTRAN, when we see an IF-statement, we don't know (without looking closely) whether it's the beginning of a leading-decision loop, the end of a trailing-decision loop, a conditional selec-

[2] The labels is a computed GOTO need not be unique.

tion, or part of a more complicated control structure (such as the mid-decision loop, discussed below). This difficulty in identifying structures makes it much harder for a reader to determine the programmer's intent.

Exercise 2-1: Use arithmetic IFs and assignment statements to accomplish the following: store +1 in S if $X > 0$, store -1 in S if $X < 0$, and store 0 in S if $X = 0$. Do the same with logical IFs. (Note that in FORTRAN '<' and '>' are written '.LT.' and '.GT.', respectively.)

Exercise 2-2: Write in FORTRAN IV a leading-decision loop that doubles N until it is greater than 100.

Exercise 2-3: Write FORTRAN IV code to compute the first Fibonacci number greater than 1000.

Exercise 2-4: Translate 'GOTO (10, 20, 30, 40),I' into IF-statements.

Exercise 2-5: Write a computed GOTO that, on the basis of a number M representing a month, branches to label 100 if the month is February, to label 200 if the month has 30 days, and to label 300 if the month has 31 days.

It is Difficult to Correlate Static and Dynamic Structures

Of course, the patterns shown above are not the only ways of combining IF and GOTO statements in FORTRAN. It is possible to write *mid-decision loops*,

```
100   ... first half of loop ...
      IF (loop done) GOTO 200
      ... second half of loop ...
      GOTO 100
200   ...
```

and much more complicated control structures. The GOTO-statement is a very primitive and powerful control structure. It is a two-edged sword because it is possible to implement almost any control regime with it—those that are good, but also those that are bad.

What makes a control regime good or bad? Mainly it is understandability; in a good control structure the static form of the structure (i.e., its appearance to the reader) corresponds in a simple way to its dynamic behavior. Therefore, it is easy for a reader to visualize the effect of a control structure from its written form. The undisciplined use of the GOTO-statement permits the construction of very intricate control structures, which are correspondingly hard to understand. We will see in Chapters 3–5 that newer languages (including FORTRAN 77) depend much less on the GOTO-statement.

The idea of a good control structure is embodied in the Structure Principle first proposed by E. W. Dijkstra[3]:

[3] See Dijkstra (1968).

> **The Structure Principle**
>
> The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations.

This principle means that it should be possible to visualize the behavior of the program easily from its written form. For example, when the execution of one segment of code precedes another in time, the statements of the first segment should precede those of the second in the program. Similarly, the statements whose execution is repeated by a loop should be easy to identify. This is simplified if they are a contiguous, indented block of text. We will see many other examples of the Structure Principle in this book.

Exercise 2-6: Suggest a practical use for mid-decision loops.

The Computed and Assigned GOTOs Are Easily Confused

Two statements in FORTRAN, the *computed* GOTO and the *assigned* GOTO, illustrate the pitfalls that await the language designer. We have seen the computed GOTO:

GOTO (L_1, L_2, ..., L_n), I

where the L_i are statement numbers and I is any integer variable. The computed GOTO transfers to statement number L_k if I contains k. Computed GOTOs are usually implemented by a jump table: The compiler stores addresses (of the statements numbered L_i) in an array in memory and then compiles code to use I as an index into this array.

FORTRAN also provides another control structure for branching to a number of different statements, the *assigned* GOTO:

GOTO N, (L_1, L_2, ..., L_n)

where N is also an integer variable. This statement transfers to the statement whose *address* is in the variable N; in other words, this is an indirect GOTO. The list of statement labels is not actually necessary since all the information necessary to perform the jump is in N. The list is provided as documentation since otherwise the reader would have no way of knowing where the GOTO goes. Most compilers do not check whether the statement whose address is in N has its label included in the list (since this check would be comparatively expensive).

The assigned GOTO must be used in conjunction with another statement, the ASSIGN statement. The effect of

```
ASSIGN 20 TO N
```

is to put the address of statement number 20 in the integer variable N. Note that this is completely different from the assignment statement `N = 20`, which stores the integer 20 into N. In general, the address of statement number 20 will not be 20 (recall that the symbolic labels in our pseudo-code had no relation to the addresses to which they were bound). Thus,

the effect of `ASSIGN 20 TO N` will be to put some other number (the address of statement 20, say 347) into N.

The computed and assigned GOTOs are easily confused; they look almost identical. Therefore, it is not uncommon for a programmer to write one where the other is expected. Let us consider the consequences of writing a computed GOTO where an assigned GOTO is intended:

```
ASSIGN 20 TO N
   :
GOTO (20, 30, 40, 50), N
```

The ASSIGN-statement will assign the address of statement number 20 (say, 347) to N. The computed GOTO will then attempt to use this as an index into the jump table (20, 30, 40, 50). In this case, the index (347) will be well out of range, but most systems do not check this so we will fetch some value out of memory to use as the destination of the jump. The result is that the program will transfer to an unpredictable location in memory, thus leading to a very-difficult-to-find bug.

Now let's consider the opposite error: using an assigned GOTO where a computed GOTO is intended.

```
I = 3
   :
GOTO I, (20, 30, 40, 50)
```

Since the assigned GOTO expects the variable I to contain the address of a statement, it will transfer to that address. In this case, it will transfer to address 3, which is almost certainly not the address of one of the statements in the list and is very likely not the address of a statement at all (low-addressed locations are often dedicated to use by the system). Again, a difficult bug results.

What are the causes of these problems? The most obvious cause is the easily confused syntax of the two constructs:

GOTO (L_1 ..., L_n), I

GOTO I, (L_1, ..., L_n)

This is a violation of the Syntactic Consistency Principle.

> **Syntactic Consistency Principle**
>
> Things that look similar should be similar and things that look different should be different.

Generally speaking, it is best to avoid syntactic forms that can be converted into other legal forms by a simple error. (FORTRAN's use of '**' for exponentiation has been criticized on this basis, since leaving out one of the asterisks converts it into a legal FORTRAN multiplication sign '*'.)

A more fundamental cause for the GOTO problem is FORTRAN's *weak typing*, which results from using integer variables to hold a number of things besides integers, such as the addresses of statements, and character strings (discussed in Section 2.4, pp. 69–70). If FORTRAN had variables of type LABEL for holding the addresses of statements, and if a LABEL variable were required in an assigned GOTO, then confusing the two GOTO statements would lead to an easy-to-find compile-time error not to an obscure run-time error. Thus, FORTRAN violates the principle of Defense in Depth.

> **Defense in Depth Principle**
>
> If an error gets through one line of defense (syntactic checking, in this case), then it should be caught by the next line of defense (type checking, in this case).

This example also illustrates one of the hardest problems of language design: identifying the *interaction* of features. (In this case, the interaction is between the syntax of the GOTOs and the decision to use integer variables to hold the addresses of statements.)

Exercise 2-7*: Explain why it is computationally expensive to check whether the destination of an assigned GOTO is in its list of statement labels.

Exercise 2-8*: Propose and defend a solution to the problem of assigned and computed GOTOs in FORTRAN. If you decide to introduce a type LABEL, keep in mind that you must analyze the interaction of this feature with others already in the language. For example, will you allow LABEL arrays? Will you allow parameters of type LABEL or allow functions to return LABELs? Discuss the pros and cons.

The DO-Loop Is More Structured than the GOTO

We have seen that the GOTO- and IF-statements provide *primitive*, or *low-level*, control structuring mechanisms. That is, they are simple components from which higher-level control structures can be built. FORTRAN contains only one built-in, higher-level control structure, the DO-loop, which we discuss next. Much like the LOOP instruction in our pseudo-code, the DO-loop provides a simple method of constructing a counted loop (sometimes called a *definite iteration*). For example,

```
      DO 100 I = 1, N
      A(I) = A(I)*2
100   CONTINUE
```

is a command to execute the statements between the DO and the corresponding CONTINUE with I taking on the values 1 through N. The variable that changes values (I in this case) is called the *controlled variable*, and the statements that are repeated, which extend from the DO to the CONTINUE with a matching label, are called the *extent* or *body* of the loop.

Exercise 2-9: Write the above example using only IF and GOTO (i.e., without the DO-loop).

Exercise 2-10: Write a DO-loop that computes into SUM the sum of the array elements A(1) + A(2) + ··· + A(100).

ing storage, it can be used for many purposes, including subversion of the type system. For example, a logical variable may be equivalenced to a real variable so that the programmer can use logical operations ('.AND.', '.OR.', '.NOT.') to access the parts of the floating-point representation of the number. This kind of programming leads to very machine-dependent, and hence nonportable, programs and is usually unnecessary anyway since there are better solutions to these problems. In many cases, these problems result from trying to use FORTRAN for applications for which it was not intended, such as string manipulation and systems programming.

The EQUIVALENCE statement has outlived its usefulness for a number of reasons. First, computer memories are now much larger so that economization of storage is no longer the problem that it was in the 1950s. Second, many computers now have *virtual memory* systems that automatically and safely manage the sharing of real memory. Finally, as we will see in later chapters, modern programming languages provide dynamic storage management facilities that also automatically and safely manage the allocation of memory.

Exercise 2-33: Suppose we have three arrays dimensioned A(1000), B(700), and C(700). Further suppose that A is never in use at the same time as B or C, but that B and C may be in use at the same time. Write an EQUIVALENCE statement to share storage as much as possible. How would you change your solution if B and C were never in use at the same time?

Exercise 2-34*: Most FORTRAN systems allow the same variables to appear in both COMMON and EQUIVALENCE statements. For example,

```
COMMON /B/ U(100), V(100)
DIMENSION W(100)
EQUIVALENCE (U(60), W(1)), (W(80), V(10))
```

What do you suppose the effects of these declarations are? You can consult a FORTRAN manual or try to figure it out on your own. What *should* be the effects of these statements?

2.6 DESIGN: SYNTACTIC STRUCTURES

Languages Are Defined by Lexics and Syntax

The *syntax* of a language is the way that words and symbols are combined to form the statements and expressions. In the previous sections, we indirectly discussed the syntax of many FORTRAN constructs. For example, we saw that a DO-loop has the word DO, followed by a statement number, followed by an integer variable, followed by the symbol '=', followed by an integer expression, followed by a comma symbol, followed by another integer expression, and optionally followed by another comma symbol and another integer expression. You have probably seen the syntax of programming languages described with a formal grammatical notation, for example,

⟨DO-loop⟩ ::= DO ⟨label⟩ ⟨var⟩ = ⟨exp⟩, ⟨exp⟩ [, ⟨exp⟩]

We will discuss these notations in Chapter 4.

The *lexics* of a language is the way in which characters (i.e., letters, digits, and other signs) are combined to form words and symbols. For example, the lexical rules of FORTRAN state that an identifier must begin with a letter and be followed by no more than five letters and digits. Lexical rules are also frequently expressed in formal grammatical notations; in fact, the term *syntax* is often used to refer to both the lexical and syntactic rules of a language. The *syntactic analysis* phase of a compiler is often broken down into two parts: the *lexical analyzer* (also called a *scanner*) and the *syntactic analyzer proper* (also called a *parser*). In the following sections, we will discuss the lexics and syntax of FORTRAN.

A Fixed Format Lexics Was Inherited from the Pseudo-Codes

The pseudo-code we developed in Chapter 1 was typical of these languages in its *fixed format* lexical conventions. It was card-oriented, that is, there was one instruction per card and particular columns of these cards were dedicated to particular purposes, for example, columns 1–4 for the operation. FORTRAN has similar lexical conventions, namely, one statement per card and columns dedicated to particular purposes:

Columns	Purpose
1–5	statement number
6	continuation
7–72	statement
73–80	sequence number

Since a statement may not entirely fit on one card, it can be continued onto following cards, if these continuation cards have a character punched in column 6. The bulk of the card, columns 7–72, was devoted to the actual statement, which was *free format*; that is, it did not have to be punched in specific columns. This general lexical style was common in languages of the late 1950s and early 1960s such as COBOL. Although it is adequate for use with punched card equipment, it is quite awkward for use with the interactive program preparation methods now generally in use. It also has other limitations that we will discuss later.

Ignoring Blanks Everywhere Is a Mistake

FORTRAN adopted the unfortunate lexical convention that blanks are ignored everywhere in the body of the statement. While this was certainly an improvement over the fixed fields of the pseudo-code interpreters, it was a significant deviation from most natural languages, in which blanks are significant.

Itisveryhardtoreadasentencewithnoblanks,

yet this is exactly what FORTRAN compilers were required to do. In FORTRAN, the statement

```
DIMENSION IN DATA (10000), RESULT (8000)
```


is exactly equivalent to

```
DIMENSIONINDATA(10000),RESULT(8000)
```

and, for that matter,

```
D I M E N S I O N IN DATA (10 000), RESULT (8 000)
```

While this may seem to be a harmless convenience, in fact it can cause serious problems for both compilers and human readers. Consider this legal FORTRAN statement:

```
DO 20 I = 1. 100
```

which looks remarkably like the DO-statement:

```
DO 20 I = 1, 100
```

In fact, it is an assignment statement of the number `1.100` to a variable called `DO20I`, which we can see by rearranging the blanks:

```
DO20I = 1.100
```

You will probably say that no programmer would ever call a variable `DO20I`, and that is correct. But suppose the programmer *intended* to type the DO-statement above but accidently typed a period instead of a comma (they are next to each other on the keyboard). The statement will have been transformed into an assignment to `DO20I`. The programmer will probably not notice the error because ',' and '.' look so much alike. In fact, there will be no clue that an error has been made because, conveniently, the variable `DO20I` will be automatically declared. According to an oft-repeated story, an American Mariner I Venus probe was lost because of precisely this error.[9]

You should also notice that the real seriousness of this problem results from the *interaction* of two language features. If it weren't for FORTRAN's implicit declaration convention, the mistake would have been diagnosed as a missing declaration of `DO20I`. This is an example of a violation of the Principle of Defense in Depth, which states that if an error can get through the first line of defense without detection, then it should be caught by the next line of defense, and so forth. FORTRAN throws out what in most languages is the most significant lexical feature: the breaks between the words. Modern programming languages have lexical conventions very much like natural languages: Blanks can be (and in some cases must be) used to separate the *tokens* (words and symbols). This is both more secure and more readable.

Exercise 2-35: Define a new set of lexical conventions for FORTRAN. They should permit blanks to be inserted for readability but avoid the problems we have discussed above.

Exercise 2-36: Suggest a new syntax for the DO-loop that is less prone to the mistake illustrated above.

[9] Like most legends, this one is part fact, part fiction. The truth seems to be that the spacecraft was destroyed when it began to behave erratically, which was caused by a missing hyphen from an assembly language program. (The story can be found in *The New York Times* for July 28, 1962 and in *Far Travelers—The Exploring Machines*, by Oran W. Nicks, NASA, 1985.) The historical details do not alter, of course, the point of the example.

The Lack of Reserved Words Is a Mistake

FORTRAN granted programmers the dubious privilege of using as variables words with meaning in FORTRAN. For example, FORTRAN permits a programmer to have an array called `IF`:

```
DIMENSION IF(100)
```

Uses of this array

```
IF (I - 1) = 1 2 3
```

are likely to be confused with `IF`-statements:

```
IF (I - 1) 1, 2, 3
```

Again, a programmer is not likely willfully to write such a deceptive statement. The point is that when a compiler has seen '`IF (I - 1)`', it still does not know whether it is processing an assignment statement or an IF-statement. The combination of ignoring all blanks and allowing keywords to be used as variables makes the syntactic analysis of FORTRAN programs a nightmare. Consider what is required to classify something as a DO-statement. The instruction may begin with the letters 'DO', a sequence of digits, a legal variable name, and an '=' sign and still not be a DO-statement; the assignment to `DO20I` is an example. Furthermore, it is even enough to see if there is a comma to the right of the equals sign since

```
DO 20 I = A(I,J)
```

is not a DO-statement.

Exercise 2-37: Classify the statement on line 5:

```
      DIMENSION FORMAT(100)
 5    FORMAT(11H) = 10*(I-J)
```

Exercise 2-38*: Describe a procedure for distinguishing between a DO-statement and an assignment statement.

Algebraic Notation Was an Important Contribution

We will not turn from lexics to syntax. Recall that one of the goals of FORTRAN was to permit programmers to use a conventional algebraic notation. This goal was partially met; for example, the expression

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

would have to be written

```
(-B + SQRT (B**2 - 4*A*C))/(2*A)
```

This is probably about the best that could be expected considering the commonly available input-output equipment at the time FORTRAN was developed (card punches and teletypes). It is interesting to note that the Laning and Zierler system, which the FORTRAN designers had seen, provided a much more natural, interactive programming notation than FORTRAN or most other programming languages, even to this day. In any case, the provision of a quasi-algebraic notation was without doubt one of the major selling points of FORTRAN.

Arithmetic Operators Have Precedence

An important idea is the *precedence*, or priority, of an operator. This was developed so that quasi-algebraic notations would be unambiguous and have the expected meaning. For example, in mathematical notation the arithmetic expression '$b^2 - 4ac$' is equivalent to '$(b^2) - (4ac)$', that is, exponentiation and multiplication are done before addition. Also, ab^2 means $a(b^2)$, so exponentiation is done before multiplication. Considerations such as these have led to the following precedences among arithmetic operators:

1. Exponentiation
2. Multiplication and division
3. Addition and subtraction

This means that in the absence of parentheses, exponentiation is done before multiplication and division, and multiplication and division are done before addition and subtraction. Operators of the same precedence (e.g., addition and subtraction) are done in order from left to right, i.e., they associate to the left:

$$a - b + c - d = ((a - b) + c) - d$$

Languages differ on the precedence of unary operators (e.g., $-b$ and $+b$); some give them a higher priority than exponentiation, others the same priority as addition. Neither is entirely consistent with mathematical convention.

Exercise 2-39: Fill in the parentheses in the following expressions:

```
A + B * C - D / E
A - B - C - D
A/B/C
A/B*C
A ** B * C
I + 1 / J - 1
```

Exercise 2-40: Compare the precedence conventions of at least three programming languages for which you can find descriptions.

A Linear Syntactic Organization Is Used

We saw in Chapter 1 that pseudo-codes were patterned after machine languages; the instructions were listed in order in exactly the same way they were stored in memory. Numeric statement labels were used that were reminiscent of the addresses of machine instructions. For the most part, FORTRAN follows this same pattern. Statements are strung together, one after another, just like the instructions in memory; they are addressed with numeric labels. This is what we mean when we say that FORTRAN has a *linear* syntactic organization; the statements are arranged in a simple sequence ('linear' = line). You are no doubt familiar with *structured* programming languages in which some statements can be nested within others, but this was a later development. The only nesting that occurs in most FORTRAN dialects is in the arithmetic expressions and in the DO-loop, although the FORTRAN 77 Standard also incorporated nested IF-statements. We will see in later chapters that *hierarchical structure* and *nesting* are important methods for conquering the complexity of large structures.

2.7 EVALUATION AND EPILOG

FORTRAN Evolved into PL/I

In this chapter we concentrated mainly on FORTRAN IV, the dialect designed in 1962 and most characteristic of first-generation languages. This should not be taken to mean that work on FORTRAN ceased in 1962; rather, it has continued to the present. For this reason we will pick up the history of FORTRAN just after the design of FORTRAN IV in 1962.

The success of FORTRAN led to its use in many applications that were not strictly scientific. This resulted in the recognition of a serious deficiency in FORTRAN—its lack of any character manipulation facilities—and to a short-lived project within IBM to design a FORTRAN V with these facilities. Later (1963) a committee was set up by SHARE (the IBM users' group) to study extending FORTRAN so that it would be useful for both commercial and scientific applications. Although the language designed by this committee was originally known as FORTRAN VI, it soon became clear that it would be impossible to satisfy all of the goals and maintain compatibility with FORTRAN. In a preliminary specification of the language released in 1964, the language was called NPL (New Programming Language), although its name was changed to PL/I in 1965 because of protests from the National Physics Laboratory in England.

As may be expected of a language that tries to be a tool for all applications, PL/I is a very large language. The number of features in PL/I and the intricacy of some of their interactions have drawn much criticism. For example, Dijkstra has said,[10] "If FORTRAN has been called an infantile disorder, then PL/I must be classified as a fatal disease." You may be surprised to hear such virulent remarks made about a programming language, although they can be understood in the context of the improvements in programming methodology taking place at that time. Specifically, the late 1960s and early 1970s saw the development of *structured programming*, a body of programming methods intended to foster easier and more reliable programming. A complex, unpredictable, large programming language was seen as more of a hindrance than a help. For this, among other reasons, PL/I's popularity has waned in recent years, although it remains in use.

[10] *A Short Introduction to the Art of Programming.*

3.3 DESIGN: NAME STRUCTURES

The Primitives Bind Names to Objects

We saw in Chapter 2 that the purpose of name structures was to organize the name space, that is, the collection of names used in the program. We also saw that in FORTRAN the primitive name structures are the declarations that define names by binding them to objects; the same is the case in Algol. There is one major difference; in FORTRAN a variable name is statically bound to a memory location, whereas in Algol we will find that a single variable may be bound to a number of different memory locations and that these bindings can change during run-time. To see why this is the case, we have to investigate the *constructors* of name structures.

The Constructors Is the Block

One of the important contributions of Algol-58 was the idea of a *compound statement*. This allows a sequence of statements to be used wherever one statement is permitted. For instance, although one statement would normally form the body of a **for**-loop, such as

```
for i := 1 step 1 until N do
  sum := sum + Data[i]
```

several statements can form the body if they were surrounded by **begin-end** brackets:

```
for i := 1 step 1 until N do
  begin
    if Data[i]>1000000 then Data[i] := 1000000;
    sum := sum + Data[i];
    Print Real (sum)
  end
```

Similarly, in Algol (as opposed to Pascal and many other languages), the body of a procedure is taken to be a single statement. For example, in the following definition of cosh the body is a single assignment statement:

```
real procedure cosh (x); real x;
  cosh := (exp(x) + exp(-x))/2;
```

The fact that a group of statements can be used anywhere that one statement is expected is an example of *regularity* in language design. Recall that the Regularity Principle tells us that a regular language is generally easier to learn and understand (other things being equal) than an irregular one. The compound statement idea had important consequences for control structures, which are discussed in Section 3.5.

Between the publication of the Algol-58 and the Algol-60 reports, much research and discussion were devoted to name structures. This included some of the problems we investigated in Chapter 2, such as the sharing of data among subprograms. The issue of name structure also interacted with other issues, such as parameter passing modes and dynamic arrays. The eventual outcome of all this work was a very important idea, *block structure*.

Blocks Define Nested Scopes

In FORTRAN we saw that environments are composed of scopes nested in two levels. All subprograms are bound in the outer (global) scope and all (subprogram-local) variables are bound in inner scopes, one for each subprogram (see Figure 2.9). Although COMMON blocks are effectively bound at the global level (since they are visible to all subprograms), in fact they must be redeclared in each subprogram. Algol-60 avoids this redeclaration by allowing the programmer to define any number of scopes nested to any depth; this is accomplished with a *block*:

```
begin declarations; statements end
```

(The only difference between a block and a compound statement is that a block contains declarations, but a compound statement does not.) This defines a scope that extends from the **begin** to the **end.** This is the scope of the names bound in the declarations immediately following the **begin**; therefore, these names are visible to all of the statements in the block. Since these statements may themselves be blocks, we can see that the scopes can be nested.

Contour diagrams are often helpful in visualizing name structures. Let's compare the program in Figure 3.1 with the contour diagram in Figure 3.2 to be sure that you understand it. Remember that the rule for contour diagrams is that we can look out of a box but we cannot look into one. Figure 3.3 shows an outline of a more complicated Algol program; its contour diagram is in Figure 3.4.

Notice that the contours are suggested by the *scoping lines* we have drawn to the left of

```
begin
  real x, y;
  real procedure cosh(x); real x;
    cosh := (exp(x) + exp(-x))/2;
  procedure f(y,z);
    integer y, z;
    begin real array A[1:y];
      :
    end
  :
  begin integer array Count [0:99];
    :
  end
  :
end
```

Figure 3.3 Nested Environments

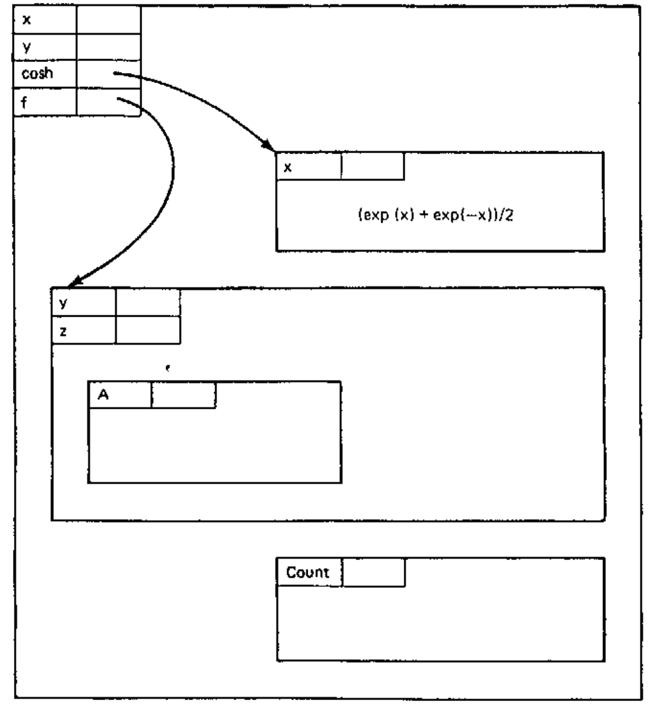

Figure 3.4 Contour Diagram of Nested Scopes

the program in Figure 3.3. Contour diagrams originated by completing scoping lines into boxes. We can see that in addition to blocks, procedure declarations also introduce a level of nesting since the formal parameters are local to the procedure. We can also see where the name "contour diagram" came from; the diagrams are suggestive of contour maps.

We have said that the purpose of name structures is to organize the name space. Why is this important? Virtually everything a programmer deals with in a program is named. Therefore, as programs become larger and larger, there will be more and more names for the programmer to keep track of, which can make understanding and maintaining the program very difficult. Another way to say this is that the *context* that programmers must keep in their heads is too large; too many names are visible. Therefore, the goal of name structures is to limit the context with which the programmer must deal at any given time. Name structures do this by restricting the visibility of names to particular parts of a program, in the case of block structure, to the block in which the name is declared. For example, in the program in Figure 3.1, the variable `val` is needed only for the two statements in the body of the first **for**-loop. Therefore, it is declared in the block that forms the body. We can see from this example that it would be very inconvenient if the variables declared in the outer blocks (`N`, `Data`, `sum`, `avg`, and `i`) were not visible in the inner block. For this reason, an inner block *implicitly inherits* access to all of the variables accessible in its immediately surrounding block; this is what is shown by the contour diagrams. The names declared in a block are called *local* to that block; those declared in surrounding blocks are called *nonlocal*. The names declared in the outermost block are called *global* because they are visible to the entire program.

Exercise 3-1: Draw a contour diagram for a program whose outline is shown in Figure 3.5. *Hint:* Recall that the body of a procedure is a single statement, which may be a compound statement or block.

```
begin integer i, j;
  procedure P(x,y); integer x, y;
    begin real z;
      :
      begin real array A[1:x];
        :
        A[i] := j;
      end
      :
      begin Boolean array B[1:y];
        :
      end
      :
    end
  procedure Q(x); real x;
    begin integer n;
    procedure R(a,b); integer a, b;
      begin integer x;
        :
      end
      :
      P(n,i);
      :
    end
  begin integer j, k;
    Q(0.0);
  end
end
```

Figure 3.5 Algol Program for Exercise

Exercise 3-2*: Before the designers of Algol decided on block structure, they considered the possibility of *explicit inheritance*, that is, having each block explicitly declare the names from the surrounding environment to which it needed access. Compare and contrast implicit and explicit inheritance and discuss some advantages and disadvantages of each. Design name structuring facilities for explicit inheritance in Algol-60.

Blocks Simplify Constructing Large Programs

You will recall that FORTRAN COMMON was designed to allow sharing data structures among a group of subprograms. One of the problems with COMMON is that the COMMON declaration must be repeated in each subprogram, which is wasteful and a potential source of errors. You have already seen that a general guideline in language design is the *Abstraction Principle*. Whenever the programmer must restate the same thing, or almost the same thing, over and over again, we should find a way to *abstract* the common parts.

To see how Algol-60 blocks apply the Abstraction Principle to shared data structures, we look again at the symbol table example from Chapter 2. Recall that we represented a symbol table as four parallel arrays called NAME, LOC, TYPE, and DIMS. These were managed by subprograms such as LOOKUP, VAR, and ARRAY1. The problem is that these subprograms needed access to the symbol table arrays but that it is undesirable to pass the arrays to them explicitly. The solution in FORTRAN was to put the arrays into a COMMON block, but this scattered about the program the information about the structure of the symbol table.

Algol block structure solves this problem since the symbol table arrays can be factored out into a block that surrounds the symbol table management procedures. This is shown in Figures 3.6 and 3.7.

Since FORTRAN COMMON blocks must be redeclared in every subprogram that uses

```
begin
  integer array Name, Loc, Type, Dims [1:100];
  procedure Lookup (n);
    ... Lookup procedure ...
  procedure Var (n, l, t);
    ... Enter variable procedure ...
  procedure Array1 (n, l, t, dim1);
    ... Enter 1-dimensional Array procedure ...
  ... other symbol table procedures ...
  ... uses of the symbol table procedures, e.g.,
  Array2 (nm, avail, intcode, m, n);
  ...
end
```

Figure 3.6 Shared Data and Block Structure

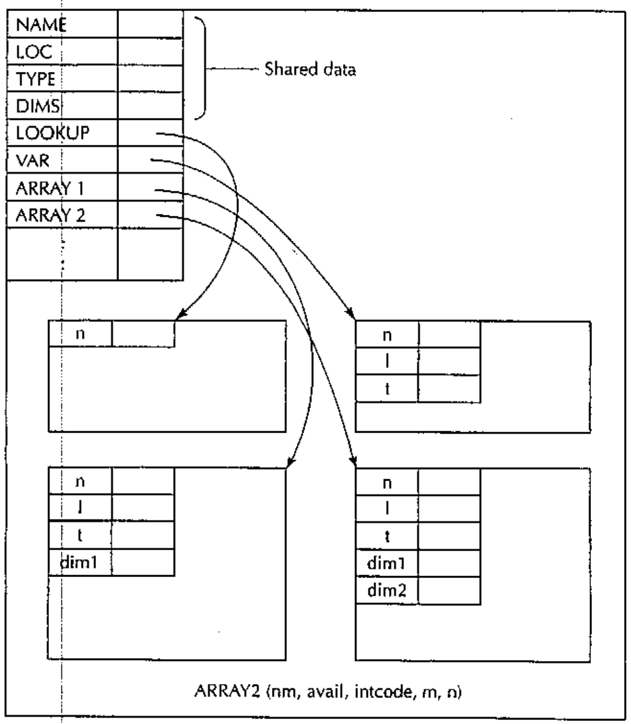

Figure 3.7 Contours Showing Shared Data

them, there is a possibility that these declarations may be mutually inconsistent, which can cause bugs that are difficult to find. In Algol the shared data structures are defined once, so there is no possibility of inconsistency. In this regard, Algol adheres to the Impossible Error Principle:

> **Impossible Error Principle**
>
> Making errors impossible to commit is preferable to detecting them after their commission.

Notice that the block that includes the declarations of the symbol table arrays must also include all the invocations (users) of the symbol table managers. Since the managers must be visible to the users, and the data structures must be visible to the managers, we can see that the data structures must be visible to the users; this is a necessary effect of Algol block structure. This means that users of the symbol table can directly access the symbol table without going through the symbol table managers. Doing so creates a maintenance problem

since the users' code will be dependent on the structure of the symbol table and will have to be modified whenever the structure of the symbol table is altered. Notice that the FORTRAN solution did not have this problem; the structure of the symbol table was confined to the COMMON block declarations, which were confined to the symbol table managers. This problem in Algol block structure is called *indiscriminate access* and was not solved for many years; its solution is discussed in Chapter 7. It is a violation of the Information Hiding Principle described in Chapter 2.

Exercise 3-3: What algebraic property of the visibility relation have we appealed to in showing that the data structures must be visible to the users?

Dynamic Scoping Allows the Context to Vary

There are two scoping rules that can be used in block-structured languages—static scoping and dynamic scoping. In static scoping a procedure is called in the environment of its *definition*; in dynamic scoping a procedure is called in the environment of its *caller*. Although Algol uses static scoping exclusively, we take this opportunity to investigate each of these scoping strategies and their consequences.

Some languages use dynamic scoping and some use static scoping. Which is better? Debate on this question dates back to at least 1960 when the advocates of Algol's static scoping confronted the advocates of LISP's dynamic scoping. To see some of the issues involved, look at this program:

```
a:begin integer m;
   procedure P;
     m := 1;
   b:begin integer m;
       P                    (*)
     end;
   P                        (**)
  end
```

With *dynamic scoping* the assignment m := 1 refers to the outer declaration of m when P is called from the outer block (**) and the inner declaration of m when P is called from the inner block (*). Look at the contour diagram in Figure 3.8 for the call (**). (In Figure 3.8 we have used DL to refer to the dynamic link, that is, to a pointer from the callee to the caller.) Since P is called in the environment of its caller, block (a), the contour for P is nested in the contour of block (a). Hence, m := 1 refers to the m declared in block (a).

The invocation (*) is represented by the contour diagram in Figure 3.9, since P is called from block (b), which is nested in block (a). We can see that the identifier m in m := 1 refers to the variable declared in block (b). What we mean when we say that P is *called in the environment of the caller* is that the contour for P is nested (dynamically) inside the contour of its caller. This is also why this scope rule is called *dynamic* nesting or *dynamic scoping*; the scope structure is determined dynamically, that is, at run-time. Thus, the context in which P is executed is the context from which it was called.

With *static scoping* the assignment m := 1 always refers to the variable m in the outer

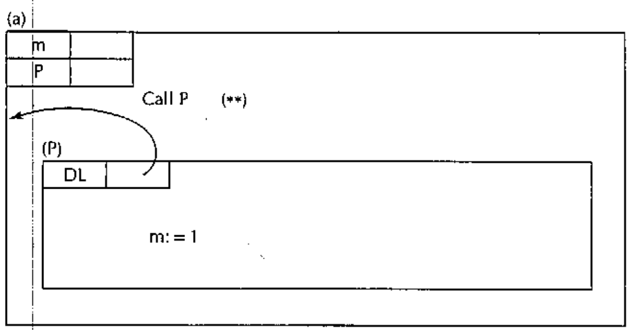

Figure 3.8 Invocation of P from Outer Block (a)

block. This is so because P is always *called in the environment of its definition*; i.e., the context in which P is executed is always the context in which it was originally defined. This means that the contour for P must be nested in the contour in which it was defined *regardless of where it is called from*. Therefore, the contour for the call (*) is as shown in Figure 3.10. Observe that the contour for P is nested in the contour for block (a) even though P was called from block (b); the context in which P executes will always be block (a) regardless of P's caller. The contour diagram shows that the m visible from the body of P is the m declared in block (a).

Since scope rules apply uniformly to all names (not just variable names), the differences between dynamic and static scoping can also be seen in the scope of procedure names. This affords a good example of the advantages and disadvantages of each.

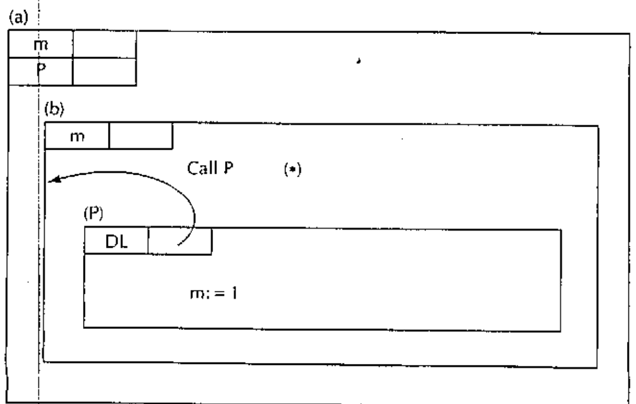

Figure 3.9 Invocation of P from Inner Block (b)

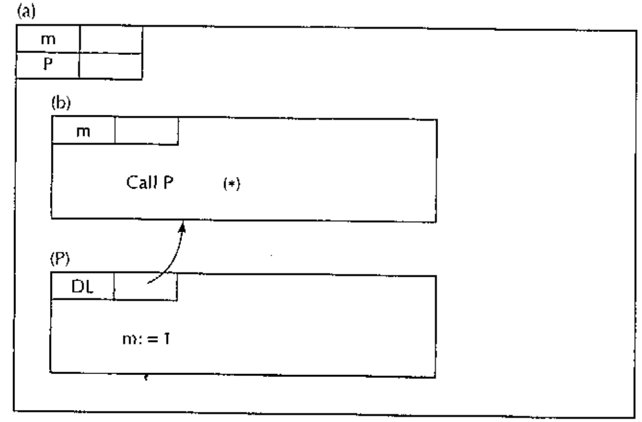

Figure 3.10 Invocation of P when called in Environment of Definition

Suppose we wished to define a function sum that summed the values of a function f from 0 to 1. This is easily accomplished with dynamic scoping:

```
begin
  real procedure sum;
    begin real S, x; S:=0; x:=0;
      for x := x + 0.01 while x≤1 do
        S := S + f(x);
      sum := S/100
    end;
    ...
end
```

To use the sum function, it is necessary only to name the function to be summed f. For example, the function $x^2 + 1$ could be summed by embedding the following block in the scope of sum (indicated by '. . .', above);

```
begin
  real procedure f(x);
    value x; real x;
    f := x↑2 + 1;
  sumf := sum
end
```

Since sum is called in the environment of the *caller*, it will be called in an environment in which f is the function $x^2 + 1$. This is one of the advantages of dynamic scoping: We can write a general procedure that makes use of variables and procedures supplied by the caller's environment. This can also be accomplished by passing these variables and procedures as explicit parameters to the procedure, which can be conveniently done in Algol with Jensen's device (described in Section 3.5).

Exercise 3-4: Show that the above definition of sum works by drawing a contour diagram for the above program when it is executing in sum.

Exercise 3-5: Write the sum procedure using Jensen's device and static scoping (see Section 3.5).

Exercise 3-6: Describe how sum would be implemented in Pascal, FORTRAN, or some other language with which you are familiar.

We have seen above how we can use to advantage the fact that in a dynamically scoped language a procedure is executed in the environment of its caller. Next, we investigate the problems to which this can lead. Suppose we wished to define a procedure `roots` to compute the roots of a quadratic equation, $ax^2 + bx + c = 0$. To do this it is useful to have an auxiliary function `discr` (a,b,c) that computes the *discriminant*, $b^2 - 4ac$. Our program could be structured like this:

```
begin
  real procedure discr (a, b, c);
    values a, b, c; real a, b, c;
    discr := b↑2 - 4 × a × c;
  procedure roots (a, b, c, r1, r2);
    value a, b, c; real a, b, c, r1, r2;
    begin
    ... d := discr (a, b, c); ...
    end
  :
  roots (c1, c2, c3, root1, root2 );
  :
end
```

Now, suppose someone happened to call our `roots` procedure from a block in which a different procedure named `discr` had been defined:

```
begin
  real procedure discr (x, y, z);
    value x, y, z; real x, y, z;
    discr := sqrt (x↑2 + y↑2 + z↑2);
  :
  roots (acoe, bcoe, ccoe, rt1, rt2);
  :
end
```


Our `discr` procedure has been inadvertently replaced by another! Needless to say, our `roots` procedure will not give the right results. In fact, if this imposter `discr` had not happened to have the right number of parameters of the right type, it would have caused our `roots` to produce an error.

Exercise 3-7: One way to decrease the probability of these errors happening is to pick "unlikely" names for our auxiliary procedures, for example, `QdiscrQ057`. Discuss the implications of this practice for program readability.

Exercise 3-8: Draw the contour diagram that illustrates the `roots` example.

The problem described above is an example of *vulnerability*, so called because the `roots` procedure is *vulnerable* to being called from an environment in which its auxiliary procedure is not accessible. To put it another way, there is no way `roots` can preserve its access to its `discr`. Vulnerability and a means of eliminating it are discussed in Chapter 7.

Exercise 3-9: Show that the above problem can be solved, even in the presence of dynamic scoping, by a proper arrangement of the nesting of `roots` and `discr`. Show, on the other hand, that if `discr` is shared by two or more procedures, then there is no way to prevent vulnerability in the presence of dynamic scoping; that is, there is no way these procedures can ensure their access to `discr`.

Static and Dynamic Scoping Summarized

Let's try to summarize what we have seen about static and dynamic scoping. In all languages the meaning of a statement or expression is determined by the context in which the statement or expression is interpreted. The context, in turn, is determined by the scope rules of the language. Since in a dynamically scoped language the scopes of names are determined dynamically, that is, at run-time, we can see that in such a language the meanings of statements and expressions may vary at run-time.

Conversely, in a statically scoped language, the scopes of names are determined statically by the structure of the program so the meanings of statements and expressions are fixed. To put this another way, the meanings of all statements and expressions can be determined by inspecting the *static structure* of the program without having to understand its *dynamic behavior.* To summarize:

- In *dynamic scoping* the meanings of statements and expressions are determined by the *dynamic structure* of the computations evolving in time.
- In *static scoping* the meanings of statements and expressions are determined by the *static structure* of the program.

Static Scoping Aids Reliable Programming

The emphasis on reliable programming in recent years has led to the general rejection of dynamic scoping. It is not hard to understand why. We know how confusing it can be if someexpecting a real. Discuss the security implications of this. How could this loophole be avoided? (Do not forget to take separately compiled subprograms into account.)

Exercise 3-19*: Identify some other safety features in the languages we have discussed and in any other languages with which you may be familiar. Propose at least two new safety features that will catch programmer errors without getting too much in the way.

3.5 DESIGN: CONTROL STRUCTURES

Primitive Computational Statements Have Changed Little

The primitives from which control structures are built are those computational statements that do not affect the flow of control. In Algol, this is the assignment statement, which is essentially the same as FORTRAN's (except that a different symbol is used ':='). Recall that in FORTRAN the input-output statements are also control structure primitives; this is not the case in Algol. Input-output is performed by library procedures rather than specialized statements. Therefore, it is quite accurate in the case of Algol to say that the function of control structures is to direct and manage the flow of control from one assignment statement to another.

Control Structures Are Generalizations of FORTRAN's

We saw in Chapter 2 that FORTRAN's control structures are closely patterned on the branch instructions of 1950s computers.[4] Algol has provided essentially the same structures in a generalized and regularized form. For example, FORTRAN has a simple logical IF-statement:

```
IF (logical expression) simple statement
```

in which the *consequent*, or **then**-part, of the IF is required to be a single, simple, unconditional statement (such as an assignment, GOTO, or CALL). In Algol this arbitrary restriction is removed, and the consequent is allowed to be any other statement, including another **if**-statement. Furthermore, the consequent is allowed to be a group of statements, as we will see.

The **if**-statement is also extended beyond FORTRAN by having an *alternate*, or **else**-part, which is executed if the condition is false, for example,

```
if T[middle] = sought then location := middle
else lower := middle + 1;
```

This allows a more *symmetric* analysis of a problem into two cases, one for which the condition is true and one for which it is false.

[4] Throughout this section 'FORTRAN' refers to 'FORTRAN IV'. We concentrate on this dialect because our intention in to contrast second-generation languages (e.g., Algol-60) with first-generation languages (e.g., FORTRAN IV).

The Algol **for**-loop is also more general than FORTRAN's DO-loop. It includes the function of a simple DO-loop, for example,

```
for 1 := 1 step 2 until N × M do
  inner[i] := outer[N × M - i];
```

It also has a variant that is similar to the **while**-loops found in languages such as Pascal. For instance, the Algol-60 **for**-loop:

```
for NewGuess := Improve(OldGuess)
    while abs(NewGuess - OldGuess) > 0.0001
  do OldGuess := NewGuess;
```

corresponds to the Pascal **while**-loop:

```
NewGuess := Improve(OldGuess);
while abs(NewGuess - OldGuess) > 0.0001 do
 begin
   OldGuess := NewGuess;
   NewGuess := Improve(OldGuess)
 end;
```

We will see later in this section that there are a number of other cases in which Algol-60's control structures are more regular, more symmetric, more powerful, and more general than FORTRAN's. There are a number of reasons for this. As we have seen, FORTRAN has many restrictions, such as the restrictions on the IF-statement mentioned above and the restrictions on array subscripts described in Chapter 2. These restrictions were made for many reasons, including efficiency (the array restrictions) and compiler simplicity (the IF restrictions). Whatever their reasons, these restrictions almost always seem inexplicable to the programmer; violations of the Zero-One-Infinity Principle and other instances of *irregularity* in a language's design make the language harder to learn and remember (the Regularity Principle). The Algol designers attempted to eliminate all asymmetry and irregularity from Algol's design. Their attitude was, "Anything that you think you ought to be able to do, you will be able to do." As we will see, they got carried away in a few instances.

Nested Statements Are Very Important

As previously mentioned, there is an irregularity in FORTRAN's IF-statement. That is, if the consequent of the IF is a single statement it can be written directly (most of the time). for example,

```
IF(X .GT. Y) X = X/2
```

But if it is more than one statement, it is necessary to negate the condition and jump over the consequent; for example,

```
      IF(X .LE. Y) GOTO 100
        X = X/2
        Y = Y + DELTA
100     ...
```

One of the most obvious problems with this is that it makes it difficult to modify a program; adding one statement to a consequent may require restructuring the entire IF-statement. Thus, the FORTRAN IF undermines *maintainability*. The other objection to FORTRAN's syntax is simply that it is irregular; conditions are written in completely different ways solely on the basis of the number of statements in their consequents. This is a source of errors since programmers may forget to negate the condition with multistatement consequents.

Recall that FORTRAN's DO-loop does not have this problem because it can be nested. That is, its syntax is

```
      DO 20 I = 1, N
        statement 1
        statement 2
           :
        statement m
20    CONTINUE
```

This allows any number of statements to be included in the body of the loop (including other DO-loops); this is clearly a much better solution. We can see that the DO and CONTINUE form *brackets*, like parentheses, that mark the beginning and end of the loop. Since FORTRAN allows CONTINUE statements to be placed anywhere in a program (they act as "do nothing" statements), matching statement labels ('20' in the example above) are used to decide which DO goes with which CONTINUE. We will see that this same approach is used in some newer programming languages.

There is another situation in which FORTRAN handles the single- and multiple-statement cases asymmetrically. As we saw, the usual way to define a function in FORTRAN is a declaration such as

```
FUNCTION F(X)
  statement 1
     :
  statement m
END
```

This is the multiple-statement case; the body of the function is bracketed by a matching FUNCTION and END (although FORTRAN does not allow functions to be nested like DO-loops). However, if the body of the function is composed of a single statement, F = *expr*, then the entire declaration can be written in an abbreviated form[5]:

```
F(X) = expr
```

Again, we can see that the two cases are handled asymmetrically.

The Algol designers realized that all control structures should be allowed to govern an arbitrary number of statements, so in Algol-58 these statements were all made *bracketing*. That is, each control structure (such as **if-then**) was considered an opening bracket that had

[5] In this case, however, the function F is local to the subprogram in which it is declared, a further instance of irregularity.

to be matched by a corresponding closing bracket (such as **end if**). Later, during the Algol-60 design, and largely as a result of seeing the BNF description of Algol-58, they realized that one bracketing construct could be used for all of these cases. The approach they used was that all control structures are defined to govern a single statement; for example, the body of a **for**-loop is a single statement and the consequent and alternative (**then**-part and **else**-part) of **if**-statements are both single statements. Even the body of a procedure is a single statement. However, the designers went on to define a special kind of statement, called a *compound statement*, that brackets any number of statements together and converts them to a single statement. That is, a group of statements surrounded by the brackets **begin** and **end**, for example,

```
begin
  statement 1;
  statement 2;
     :
  statement n
end
```

is considered a single statement and can be used anywhere a single statement is allowed. Look again at Figure 3.1 and compare the two **for**-loops. The body of the first is a compound statement containing two simple statements, and the body of the second is a single assignment statement. In Figure 3.3 we can see several procedure declarations. In the definition of cosh the body of the procedure is a single simple statement, similar to FORTRAN's abbreviated function definition. The declaration of f is the more common case, in which the body of the procedure is a compound statement.

You have probably noticed by now that in Algol the **begin–end** brackets do double duty—they are used both to group statements into compound statements and to delimit *blocks*, which define nested scopes. This is a lack of *orthogonality* in Algol's design; there are two independent functions—the defining of a scope and the grouping of statements—that are accomplished by the same construct. This may seem like an economy, but it often leads to problems. For example, we saw in Section 3.2 that block entry requires the creation of an activation record to hold the local variables of the block. Since in a compound statement there are no local variables, no activation record is required. In fact, it would be quite inefficient and needless to create an activation record everywhere a compound statement is used since this is just a syntactic mechanism for grouping statements together, similar to parentheses in expressions. Therefore, it is necessary for compilers to determine whether any variables or procedures are declared in a block or procedure in order to determine whether or not to generate block entry-exit code. We will see in later chapters that newer languages have separated the two functions of statement grouping and scope definition.

We should point out that Algol's syntax does not entirely solve the problems with FORTRAN's syntax. In particular, there still is a minor maintenance problem since if a loop body (or procedure body, or consequent, or alternate) is changed from a single statement to several statements, the programmer must remember to insert the **begin** and **end.** Forgetting this is a common mistake, since their absence is not obvious in a well-indented program: for example,

```
for i := 1 step 1 until N do
      ReadReal(val);
      Data[i] := if val < 0 then -val else val;
for 1 := 1 step ...
```

For this reason many Algol programmers have adopted the *coding convention* of always using **begin** and **end**, even if they surround only a single statement. We will see in Chapter 8 that newer languages have solved this problem.

Exercise 3-20*: We have seen several problems with blocks and compound statements in Algol. Discuss some alternate approaches that have the advantages of blocks and compound statements but solve these problems.

Exercise 3-21*: In FORTRAN, a CONTINUE matches a DO-loop only with the same statement number, whereas in Algol an **end** matches the nearest preceding unmatched **begin**. Furthermore, the same brackets are used for all nested statements. Discuss the consequences of a missing **end** in the middle of a large, deeply nested Algol program. When is the compiler likely to notice the error? What sort of diagnostic would it produce? Suggest improvements.

Compound Statements Are Hierarchical Structures

Algol's compound statement is a good example of *hierarchical structure*. Starting with the simple statements, such as assignment, procedure invocation, and the **goto**-statement, complex structures are built up by *hierarchically* combining these statements into larger and larger compound statements. Hierarchical structure is one of the most important principles of language and program design.

Nesting Led to Structured Programming

We saw that in FORTRAN an IF-statement with a compound consequent had to be implemented with a GOTO-statement; specifically, the GOTO-statement was used to skip the statements of the consequent. Algol eliminated the need for the **goto**-statement by using compound statements. The same situation arises in an **if-then-else** statement. The Algol code

```
if condition then
   begin
     statement 1;
        :
     statement m
   end
else
   begin
     statement 1;
        :
     statement n
   end
```


can be expressed in FORTRAN only by a circumlocution:

```
      IF (.NOT.(condition)) GOTO 100
        statement 1
            :
        statement m
        GOTO 200
100     statement 1
            :
        statement n
200     ...
```

As we said in Chapter 2, the GOTO is the workhorse of control-flow in FORTRAN.

Almost as soon as programmers began writing in Algol-60, they noticed that many fewer **goto**-statements were required than in other languages. They also noticed that their programs were, on the whole, much easier to read. This led several computer scientists, including Peter Naur, Edsger Dijkstra, and Peter Landin, to experiment with programming without the use of **goto**-statements. This is completely impossible in a language like FORTRAN, but it is quite possible in Algol. It prompted Edsger Dijkstra to write, in 1968, a now-famous letter to the editor of the *Communications of the ACM*. It was called "Go To Statement Considered Harmful" and stated that "the **go to** statement should be abolished from all 'higher level' programming languages." Dijkstra discovered that the difficulty in understanding programs that made heavy use of **goto**-statements was a result of the "conceptual gap" between the static structure of the program (spread out on the page) and the dynamic structure of the corresponding computations (spread out in time). We call this the Structure Principle:

> **The Structure Principle**
>
> The static structure of the program should correspond in a simply way to the dynamic structure of the corresponding computations.

Dijkstra's letter sparked immediate and vigorous debate and by 1972 led to an entire session of the ACM National Conference being devoted to the "go to controversy." Much of it reflects "fascination and fear" of the ampliative and reductive aspects of **goto**-less programming (recall Section 1.4). Ultimately this led to a loose body of programming methods and techniques called *structured programming* and a greater awareness among computer programmers and scientists of the problems of reliable programming. Although most of the controversy has now died down and most programming languages still have a **goto**-statement, these statements are needed much less often. Most programming languages have a rich set of *structured* control structures and most programmers have a better understanding of when a **goto**-statement is appropriate. This is only one example of a situation in which a programming language issue has been the focal point of a wider issue in programming methodology.

less likely to have dangerous interactions. Other examples of feature interactions will be discussed later in this book.

The for-Loop Is Very General

Earlier in this chapter, we looked at the Algol **for**-loop, which is a generalization of FORTRAN's DO-loop. We saw the following two forms:

```
for var := exp step exp' until exp" do stat
for var := exp while exp' do stat
```

In fact, the Algol **for**-loop is much more general than this. The idea behind it is that the **for**-loop generates a sequence of values to be assigned successively to the controlled variable. This sequence of values is described by a list of *for-list-elements*. For example, the for-list-element

```
1 step 1 until 5
```

generates the sequence 1, 2, 3, 4, 5. Also, if the most recent value of `i` were 16, the for-list-element

```
i/2 while i ≥ 1
```

would generate the values 8, 4, 2, 1. Finally, values to be used in the for-list can be listed explicitly; for example,

```
for days := 31, 28, 31, 30, 31, 30,
            31, 31, 30, 31, 30, 31, do ...
```

causes the controlled variable to take on the values of the days in the months. Algol even permits a conditional expression to be used, thus allowing leap years to be handled correctly[6]:

```
for days := 31,
  if mod (year, 4) = 0 then 29 else 28,
  31, 30, 31, 30, 31, 31, 30, 31, 30, 31 do ...
```

Exercise 3-28: The **for**-loop above is not quite correct, since a year is not a leap year if it is divisible by 100 (but not 400). Modify the **for**-loop to handle this case correctly.

Exercise 3-29: Suppose Algol did not have the ability described above for listing arbitrary sequences of values in a for-list (most languages do not). How would you solve the problem of programing the above loop? Remember to handle leap years correctly.

[6] We have assumed a mod function, which is not a built-in function in Algol-60.

The for-Loop Is Baroque

You may be surprised at the generality of the Algol **for**-loop, but it has even greater possibilities! The sequence of controlled variable values can be defined by a list of for-list-elements. For example, the **for**-loop

```
for i := 3, 7,
         11 step 1 until 16,
         i/2 while i ≥ 1,
         2 step i until 32
do print (i)
```

will print the sequence of values

```
3 7 11 12 13 14 15 16 8 4 2 1 2 4 8 16 32
```

Check this to be sure you understand it.

There is another aspect of Algol's **for**-loop that deserves mention—the binding time of the loop parameters. Algol specifies that any expressions in the current for-list-element are reevaluated on each iteration (as is used in '**step** i' in the preceding example). For example, the body of a **for**-loop such as

```
for i := m step n until k do ...
```

may change the values of the loop parameters `i`, `m`, `n`, and `k`. Of course, this means that the programmer really does not have a very clear idea of how many times the loop will iterate. This undermines the idea of a *definite iteration*, which is what the **for**-loop is supposed to be. Furthermore, these expressions must be reevaluated on each iteration even if they have not changed. This means that the most common loops, which do not change their parameters during the loop, will bear the cost of providing for these more general loops. This violates a basic principle of language design.

The Localized Cost Principle

Users should pay only for what they use; avoid distributed costs.

In other words, language designers should avoid features whose costs are distributed over all programs regardless of whether these features are used.

It is difficult to imagine why anyone would ever need this much generality in a **for**-loop. Perhaps a fascination with technological possibilities created a tendency to this extrapolation. Computer scientists call constructs such as this, which are cluttered with features of extreme generality and doubtful utility, *baroque*. As you are probably aware, this term refers to a style of art characterized by elaborate and rich ornamentation and a certain irregularity in shape. Language designers call a language baroque when it is irregular in structure and has a surplus of features ("decoration") of questionable usefulness. Baroque became a pejorative term during the movement toward simplicity that characterized the third-generation languages (including Pascal). This is discussed in Chapter 5.

require only 5 bits (since there are only five weekdays). If there are 256 characters in the character set, then a **set of char** will require 256 bits.

We have seen that sets are represented very compactly. How efficient are the set operations? It turns out that the set operations are simple to implement and very fast. Consider set intersection: We want an element to be in the intersection only if it is in both of the operand sets. In other words, we want the element's bit to be set in the result set only if it is set in both of the operand sets. This is just an 'and' operation on the bit strings. This example shows how doing a bit-by-bit 'and' between `S` and `T` gives the bits for `S * T`:

```
S   =  1  1  1  0  1  0  1  0  0  0
T   =  1  1  1  1  1  1  0  0  0  0
S*T =  1  1  1  0  1  0  0  0  0  0
```

You are probably aware that most computers have instructions for performing a logical 'and' between bit strings; in fact, these are often the fastest operations a computer can do, faster even than integer arithmetic. Of course, if the set takes more bits than will fit in a word, then several 'and's may be required. For example, with a 32-bit word size, it will take eight (256/32) 'and's to intersect two sets of characters. This is still quite efficient.

The other set operations are also simple: Union is a logical 'or', complementation is a logical 'not', and equality and inequality tests are simple comparisons of the bit strings. Testing whether a given value is a member of the set, `x in S`, reduces to determining if the bit corresponding to x is set. On most machines this can be accomplished by shifting this bit into the most significant position and doing a sign test. Again, this is very efficient. A subset test is performed by an 'and' and equality test since

$$S <= T \text{ if and only if } S = S * T.$$

The other relations are implemented similarly.

In summary, the set-type constructor is almost ideal: It is very high level, very readable, and efficiently implemented. It also enhances security because it does all the normal type checking and it saves programmers from having to do their own error-prone bit manipulation. The set type illustrates

> **The Elegance Principle**
>
> Confine your attention to designs that *look* good because they *are* good.

Exercise 5-7: Write a type declaration for sets of days of the week.

Exercise 5-8: Write a test, using set operations, for determining if the character in `ch` is *not* alphanumeric.

Array Types

Algol-60 generalizes FORTRAN arrays in two respects: It allows any number of dimensions and it allows lower bounds other than one. We will see that Pascal has generalized Algol's arrays in some respects and has restricted them in others.

One of the generalizations is in the allowable *index types*. In FORTRAN and Algol, arrays could be subscripted only by integers; in Pascal they can be subscripted by many other types, including characters, enumeration types, and subranges of these. This is simple to understand. Suppose we declare an array `A`, holding 100 reals, indexed by the integers 1 to 100:

```
var A: array [1 .. 100] of real;
```

Notice that the dimensions of the array have been specified as a *subrange* of the integers.

Now, suppose we wanted an array to record the number of hours worked on each of the days Monday through Friday. We could declare an array with the dimensions `1 .. 5`, but we have already discussed the disadvantages of manually encoding things as integers. A better approach is to use a subrange of `DayOfWeek` as the index type:

```
var HoursWorked: array [Mon .. Fri] of 0 .. 24;
```

Think of this as a table whose entries are labeled with `Mon`, `Tue`, ... , `Fri`:

	Hours
Mon	
Tue	
Wed	
Thu	
Fri	

Notice that we have also made the *base type* of the array `0 .. 24` since we know that it is impossible to work fewer than zero or more than 24 hours in a day. This adds security to our program.

Arrays subscripted by noninteger index types can be used in the usual way. For example, to find the total number of hours worked in the week:

```
var day: Mon .. Fri;
    TotalHours: 0 .. 120;
begin
  TotalHours := 0;
  for day := Mon to Fri do
    TotalHours := TotalHours + HoursWorked[day];
```

Notice that it is not necessary to check the bounds of `HoursWorked` at run-time; since `day` is of type `Mon .. Fri` it *cannot* hold an illegal subscript value. This is one of the advantages of using subranges: Much checking can be done at compile-time rather than run-time.

Actually, any *finite discrete type* (i.e., any type that can be represented as a finite contiguous subset of the integers) can be used as an index type. For example, if we wanted to count the number of occurrences of different characters, we could declare

```
var Occur: array [char] of integer;
```

Then, `Occur[ch] := Occur[ch] + 1` would increment the number of occurrences of the character in `ch`. We could test if there are more 'e's than 't's by

```
if Occur['e'] > Occur['t'] then ....
```

Another way in which Pascal generalizes Algol arrays is in the allowable element types. In Algol the programmer is allowed to have arrays of reals, integers, or Booleans (since these are the only data types in the language). In Pascal any other type can be the *base type* of an array type. That is, we can have arrays of integers, reals, characters, enumeration types, subranges, records, pointers, and so forth.

In general, a Pascal array-type constructor has the form

```
array [⟨index type⟩] of ⟨base type⟩
```

where ⟨index type⟩ is any finite discrete type and ⟨base type⟩ is any type at all. Thus, Pascal arrays can be considered *finite mappings* from the index type to the base type.

So far we have discussed only one-dimensional arrays. Does Pascal allow multidimensional arrays? In fact, it does not, although the other generalizations of Pascal more than compensate for their absence. Suppose we need a 20 × 100 array of reals M; this can be considered a 20-element array, each of whose elements is a 100-element array of reals. That is,

```
var M: array [1 .. 20] of array [1 .. 100] of real;
```

As we said, the base type of an array can be any type, including another array type.

Subscripts can be combined to access any element of the matrix. For example, since `M[3]` is the third row of M, `M[3][5]` is the fifth element of the third row of M; in other words, $M_{3,5}$. Pascal allows the programmer to use more standard notation by providing `M[i,j]` as *syntactic sugar* for `M[i][j]`. Similarly, the declaration of M can be written

```
var M: array [1 .. 20, 1 .. 100] of real;
```

although it is still interpreted as an array of arrays. Thus, although the programmer has lost neither power nor convenience, the language and the compiler have been simplified because they have to deal only with one-dimensional arrays.

Problems with Array Bounds

There are two significant ways in which Pascal's arrays are more restrictive than Algol's. Recall that Algol has dynamic arrays: The bounds of the array are computed at scope entry time and can vary from one activation of the scope to another. This is not the case in Pascal; all arrays are static just as in FORTRAN. Why was this useful, efficient facility deleted? One reason was that dynamic arrays are a little less efficient than static arrays, but this does not seem to be the primary reason. Rather, static arrays are implied by two fundamental design decisions in Pascal:

1. All types must be determinable at compile-time.
2. The dimensions are part of an array type.

The first decision is required in order to be able to do type checking at compile-time. The result is that all Pascal objects have *static types*; they cannot change at run-time. The

second decision means that two array types are considered the same if their index types match and their base types match. This is usually reasonable; it does not make much sense to assign a `1..100` array of reals to a `-20..20` array of reals. Notice, however, that these two decisions *interact* to imply static arrays: Since the dimensions are part of the type and the type must be static, the dimensions also must be static. In this case, the *feature interaction* is not too serious; we can live without dynamic arrays, particularly in a language intended for teaching. Next we will discuss a more serious example of feature interaction.

Consider these two Pascal[2] design decisions (we have already discussed the first):

1. The dimensions are part of an array type.
2. Pascal enforces strong typing; therefore, types of actuals must agree with types of formals.

Generally, *strong typing* says that in any context in which a thing is used, the type of that thing must agree with the type expected in that context. In particular, the types of actual parameters must agree with the types of the corresponding formal parameters. Since the dimensions of an array are part of its type, this means that the dimensions of an actual array parameter must agree with the dimensions of the corresponding formal array parameter. Let's look at an example.

```
type vector = array [1 .. 100] of real;
var U, V: vector;
function sum (x: vector): real;
    ...
begin ... end {sum};
```

Given these definitions, it is perfectly legal to write `sum(U)` and `sum(V)` since the types of U and V match the type of x. Suppose we have another array W, of length 75:

```
var W: array [1 .. 75] of real;
```

It is not legal to write `sum(W)` because the types of W and x don't agree. If we want to sum the elements of W, we will have to write another sum procedure that works on 75-element arrays! In fact, our sum procedure will not even work on 100-element arrays whose index type is `0..99`! We will have to write a separate sum procedure for every different length array that appears in our program (and that we want to sum).

This situation is a terrible state of affairs; it is a gross violation of the Abstraction Principle. Furthermore, it makes Pascal almost unusable for programs that perform similar manipulations on a large number of different size arrays, such as scientific programs. It is impossible to write a general array manipulation procedure in Pascal. We can see that this results from the *interaction* of two design decisions that separately seem quite reasonable. Anticipating undesirable feature interactions is one of the most difficult aspects of language design.

We will see in Chapter 7 that Ada has eliminated this problem, as well as restored dy-

[2] We mean the Pascal of the Revised Report (Jensen and Wirth, 1973). Although the problem has been corrected in the ISO Standard, it is still a good illustration of feature interaction.

namic arrays, by changing the decision on which both restrictions are predicated: Dimensions are not considered part of an array type in Ada. The Pascal standardization efforts have solved the array parameter problem by defining a *conformant array schema* that can be used to specify a formal parameter. Thus, if we write the header for sum as follows:

```
procedure sum (x: array [lwb .. upb: integer] of real): real;
```

then any real array indexed by integers can be passed to sum. The calls sum(U), sum(V), and sum(W) are all legal. On each call of sum the identifiers lwb and upb are bound to the lower and upper bounds, respectively, of the index type of the actual corresponding to x. A typical use of these identifiers would in the limits of a **for**-loop:

```
for i := lwb to upb do
  total := total + x[i];
```

The syntax of a simple conformant array schema is

array [⟨identifier⟩ .. ⟨identifier⟩ : ⟨type identifier⟩] **of** ⟨type identifier⟩

Although conformant array schemas solve the array parameter problem, they do so at the expense of extra language complexity.

- ***Exercise 5-9*:*** Discuss conformant array schemas as a solution to the array parameter problem. Do you think the right decision was made? Can you suggest an alternative?
- ***Exercise 5-10*:*** Suppose strings are represented as arrays of characters. What problems would you face upon implementing a string manipulation package in Revised Report Pascal (i.e., Pascal without conformant array schemas)? Discuss possible solutions.
- ***Exercise 5-11*:*** Write the procedure headers for a string manipulation package for ISO Standard Pascal (i.e., Pascal with conformant array schemas).
- ***Exercise 5-12:*** Write a set of Pascal procedures for performing mathematical operations (sum, inner product, length, etc.) on real vectors of any dimension.
- ***Exercise 5-13*:*** The ISO Pascal Standard defines two "levels of compliance." A compiler complies at level 1 if it implements all of the standard *except* conformant array schemas; it complies at level 2 if it implements all of the standard *including* conformant array schemas. Why do you suppose there are these two levels of compliance? Discuss the pros and cons.

Record Types Group Heterogeneous Data

One of the most important data structure constructors provided by Pascal is the *record-type constructor.* This is a data structure that allows arbitrary groupings of data. The idea first appeared in commercial data-processing languages such as COBOL; in the mid-1960s Hoare suggested adding the facility to scientific languages. Records appeared in both Algol-W and several extensible languages.

A typical example of a record, a personnel record, appears in Figure 5.3. Just like an array, a record has a number of *components*. Unlike an array, however, the components of a


```
type person =
  record
    name: string;
    age: 16 .. 100;
    salary: 10000 .. 100000;
    sex: (male, female);
    birthdate: date;
    hiredate: date;
  end;
string = packed array [1 .. 30] of char;
date = record
    mon: month;
    day: 1 .. 31;
    year: 1900 .. 2100;
  end;
month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
         Nov, Dec);
```

Figure 5.3 Example of a Record Type—A Personnel Record

record can be of different types, as we can see in the definition of person. Notice also that the components of records can themselves be complex data types. For example, the name component is a string, which is a 30-element array of characters. Hence, records can contain arrays. Also, the birthdate and hiredate components have the type date, which is itself a record with three components. Hence, records can contain other records.

The components of arrays are selected by subscripting. How are the components of records selected? Suppose we have declared newhire to be a variable of type person:

```
var newhire: person;
```

A component of a record is selected by placing a period between the name of the record and the name of the component. For example, to set newhire's age and sex we can write

```
newhire.age := 25;
newhire.sex := female;
```

If today is a variable of type date, then newhire's hiredate can be set to today by

```
newhire.hiredate := today;
```

Notice that this is an assignment of one record variable to another. This is legal since the record name denotes the entire record, which can be assigned and compared for equality—like variables of any other types.

Selectors for records and arrays can be combined as needed to access a particular component. For example, since newhire.hiredate is itself the name of a record, we can use the dot notation to select its components. To set the date of hire to June 1, we can write

```
newhire.hiredate.mon  := Jun;
newhire.hiredate.day  := 1;
```

Similarly, we can test whether the first character of `newhire`'s name is an 'A' by

```
if newhire.name[1] = 'A' then ....
```

As another example, we can use an array to hold the personnel records of all our employees:

```
type employeeNum = 1000 .. 9999;
var employees: array [employeeNum] of person;
     EN: employeeNum;
```

If we now wish to get the year of birth of the employee whose number is in EN, we can write

```
employees[EN].birthdate.year
```

Notice that `[EN].birthdate.year` essentially defines an *access path* to a particular component of the `employees` data structure.

Observe that a record type is a *scope-defining* structure in Pascal; it groups the field names together. The field names are not visible outside of the record declaration unless a record is "opened up" with the dot operator.

If a number of successive statements reference fields of one record, such as

```
newhire.age  := 25;
newhire.sex  := female;
newhire.salary := 30000;
```

then Pascal permits this record to be opened once for all of them. This is accomplished by the **with**-statement:

```
with newhire do
 begin
    age  := 25;
    sex  := female;
    salary := 30000
 end;
```

This ability to *enter* another environment and make its names visible becomes very important in the fourth-generation languages discussed in Chapter 7.

You may have wondered why Pascal includes both arrays and records since they are both methods of grouping data together. They differ in two important respects. Arrays are *homogeneous* (homo = same; genus = kind), that is, all of the components of an array are the same type; records are *heterogeneous* (hetero = different), that is, their components do not have to be the same type (consider `person`). In this sense records are more general than arrays.

The other difference between arrays and records is in their manner of selecting components. We can select specific array elements with expressions like `A[1]`, `A[2]` just as we can select specific record components with expressions like `R.mon`, `R.day`. The difference is that we can compute the selector to be used with arrays; that is, we can write `A[E]` where

E is an expression whose value will be known only at run-time. This is an important feature since it allows, for example, writing a loop that processes all the elements of an array. This cannot be done with records; we cannot write an expression like `R.`*E*, where *E* may refer to any of the fields `mon`, `day`, or `year`. Why? Recall that it is a basic design decision of Pascal that all types can be checked at compile-time (i.e., that Pascal is *statically* typed). Since all of an array's elements are the same type, the compiler knows the type of `A[`*E*`]` even if it does not know which element will be selected. This is not the case with records: `R.`*E* has different types, depending on whether *E* is `mon`, `day`, or `year`. The differences between arrays and records are summarized in the following table:

Composite Data Structures

	Element types	Selectors
Array	homogeneous (less general)	dynamic (more general)
Record	heterogeneous (more general)	static (less general)

■ ***Exercise 5-14:*** Define a record type `automobile`, including all auxiliary declarations, for an application such as an automobile registration database. Automobiles should have attributes such as `manufacturer`, `model`, `year`, `value`, `owner`, and so forth.

Variant Records Allow Alternative Structures

Next, we discuss another kind of record: *variant records.* To see their motivation, suppose that we are writing a program to keep track of all of the airplanes of an airline company. What data structure should we use? A record is the natural choice since each plane will have a number of different attributes and these will surely be of different types. What are some of these attributes? For all planes we will want to know the flight number and the type of aircraft. If a plane is in the air, then it will have an *altitude, heading, arrival time,* and *destination.* If it is landing or taking off, then it will have an *airport* and *runway number.* If it is parked at a gate at the terminal, then it will have an *airport, gate number,* and *departure time.* Possible type declarations are shown in Figure 5.4.

There are at least two problems with this approach. First, it is inefficient. A `plane` record will have to contain space for all of these fields, even though some of them cannot be in use at the same time. For example, since a plane cannot be in the air and parked at a gate at the same time, it cannot have an altitude and a gate number at the same time. It would be helpful if, like Algol blocks, there were some way to declare disjoint subrecords that could not be in use at the same time.

There is also a potential *security* problem in this record definition. It allows us to perform meaningless operations, such as to ask the altitude of a plane whose status is `onGround`. What we need is some way of grouping the different fields according to the status with which they are associated. This is the function of a *variant record.*

The situation with which we are faced is the following: Certain attributes are possessed by all planes, regardless of their status (i.e., flight number and kind of aircraft). Other at-

```
type plane = record
   flight:        0 .. 999;           {flight number}
   kind:          (B727, B737, B747);
   status:        (inAir, onGround, atTerminal);
   altitude:      0 .. 100000;        {feet}
   heading:       0 .. 359;           {degrees}
   arrival:       time;
   destination:   airport;
   location:      airport;
   runway:        runwayNumber;
   parked:        airport;
   gate:          1 .. 100;           {gate number}
   departure:     time
end {plane};

time = packed record hrs: 00 .. 23; min: 00 .. 59 end;
airport = packed array [1 .. 3] of char;
runwayNumber = packed record dir: (N,E,S,W); num: 00 .. 99
   end;
```

Figure 5.4 Record without Variants

tributes have meaning only when the plane is in a certain status. What we need is a record type with a *variant* for each possible situation in which a plane can be. Such a *variant record* is illustrated in Figure 5.5.

The status of a plane at a given time is indicated by the value of the *tag field*, `status`. Since the status of a plane can be only one of `(inAir, onGround, atTerminal)` at a time, the fields in different variants cannot be in use at the same time. This means that, just like disjoint blocks in Algol, they can share the same storage locations.[3] This solves the efficiency problem: Since only one of the variants can exist at a time, the compiler need set aside storage only for the largest variant. This is illustrated in Figure 5.6. In this case, the `inAir` variant requires the most space.

Variant records also solve the *security* problem that we discussed. Since the *altitude* field has meaning only when the plane's status is `inAir`, the compiler can generate code to check that the tag field has the correct value before permitting a field reference.

Unfortunately, variant records introduce a loophole into Pascal's type system. Pascal does not require the programmer to initialize the fields of a variant after changing the value of the tag field.[4] The values found in these locations will be whatever was left there from the previous variant and may be of a different type. This lack of transparency has actually

[3] The analogy with blocks extends further, since variant parts can contain variant parts. That is, variant parts, like blocks, can be nested.

[4] This is in reality an uninitialized storage problem. Pascal, like most block-structured languages, does not require variables to be initialized when their scope is entered. Access to uninitialized variables may yield the values left from the block previously using the same area of storage.


```
type plane = record
   flight:        0 .. 999;
   kind:          (B727, B737, B747);
   case status:   (inAir, onGround, atTerminal) of
   inAir: (
      altitude:     0 .. 100000;
      heading:      0 .. 359;
      arrival:      time;
      destination:  airport);
   onGround: (
      location:     airport;
      runway:       runwayNumber);
   atTerminal: (
      parked:       airport;
      gate:         1 .. 100;
      departure:    time)
end {plane};
```

Figure 5.5 Type Declaration for Record with Variants

been used intentionally as a means of getting around the Pascal type system! As Wirth remarks, "the variant record became a favorite feature to breach the type system by all programmers in love with tricks, which usually turn into pitfalls and calamities."[5] We can see that the root cause of the problem is that variant records permit a form of *aliasing* since the fields in the different variants are aliases for the same memory locations. In Chapter 7 we will see the way that Ada has solved this problem.

Pascal's Data Structures Exhibit Responsible Design

Pascal's data structures, which derive largely from the work of C. A. R. Hoare, are good examples of responsible programming language design; recall:

> **The Responsible Design Principle**
>
> Do not ask users what they want; find out what they need.

When Pascal was designed in the late 1960s (and still, in some quarters), programmers believed that they needed machine-level logical operations ('and's, 'or's, shifts, etc.) for manipulating data fields packed into single machine words. It would, of course, have been easy to include these operations in Pascal, but instead of doing so, Pascal provides a solution to

[5] N. Wirth, "Recollections About the Development of Pascal." *SIGPLAN Notices 28*, 3 (1993), pp. 333–342.

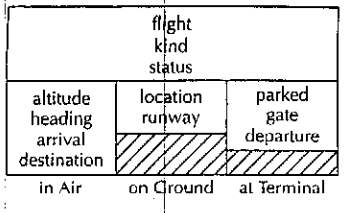

Figure 5.6 Memory Layout of a Variant Record

the problem: packing information compactly. A number of Pascal data structures, including enumeration types, subranges, finite sets and packed arrays and records, together allow a high-level, application-oriented description of data that permits just as compact representations as could be achieved in assembly language or a low-level machine-oriented language. (This ability is also aided by the data types' adherence to the Preservation of Information Principle.)

Pointer Types Are Secure

You are probably well aware of the value of linked lists, trees, graphs, and other linked data structures in many applications. These all make use of *pointers*, the ability to have one memory location contains the address of another location (or block of locations). For many years most programming languages did not provide any way for programmers to use pointers; programmers had to program in assembly language to make effective use of them. The reason is that the early high-level languages (e.g., FORTRAN and Algol) were designed for scientific programming, and linked structures were not often needed in this application area. Linked structures were most often needed in systems programming, which was almost always done in assembly language.

In the late 1960s and early 1970s, many programmers began to realize the advantages of doing *all* programming, including systems programming, in higher-level languages. This led to a demand for a pointer facility to permit machine addresses to be manipulated through higher-level languages.

Some languages (e.g., PL/I) have satisfied this demand by introducing a single, new primitive type, called, for example, **pointer**. Here is an example using this feature (it is not in a real language):

```
var    p: pointer;
       x: integer;
begin
  new(p);
      :
  p↑  := 5;
      :
  x  := x + p↑;
      :
end
```


This program allocates a memory location and puts its address in p, stores 5 in the memory location whose address is in p (that is the meaning of p↑), and then adds the contents of this location to x.

Unfortunately, this approach is not compatible with *strong typing*. Consider this example:

```
var    p: pointer;
       x: real;
       c: char;
begin
  new(p);
  p↑  := 3.14159;
  c  := p↑;
end
```

This stores a real number in the location pointed to by p and then moves the contents of this location to the character variable c. We have subverted the type system; we have managed to get a real number into a character variable by going through the (untyped) pointer p. The problem is that the system has no way of knowing the type of the memory location whose pointer is in p.

Although programmers do sometimes subvert the type system intentionally, this kind of error usually happens accidently. In programs that do a lot of pointer manipulation, it is not unusual for a programmer to think that a pointer points to something other than what it actually points to. This is the reason that Pascal provides *typed* pointers (also called *pointer binding*); these allow the compiler to enforce strong typing even when pointers are used. In Pascal a pointer is the address of an object of a particular type. Thus, there is not one pointer type but a constructor for creating many pointer types. This is written

↑ ⟨type name⟩

This is the type of all pointers to things of type ⟨type name⟩. For instance, our previous example would be written this way in Pascal:

```
var    p: ↑real;
       x: real;
       c: char;
begin
  new(p);
  p↑  := 3.14159;
  c  := p↑;      {Illegal!}
end
```

The assignment to c is now illegal because (1) the type of p is pointer to **real**, (2) thus p↑ is the name of a **real** variable. (3) Since it violates strong typing to assign a **real** variable to a character variable, (4) it is thus illegal to assign p↑ to c. Typed pointers eliminate the *possibility* of many of the bugs that plague programs in both assembly languages and high-level languages with untyped pointers. Thus Pascal pointer types obey the Impossible Error Principle, since certain runtime errors have been made impossible.

The *base type* of a pointer type can be any other type, including records and arrays. This means that the pointer following operator (i.e., '↑') can form a part of an access path along with array and record selectors. For example, if we have declared

```
var p: ↑plane;
```

then we can access the first character of the airport at which the plane is parked by

```
p↑.parked[1]
```

The elements of records and arrays can be pointers, so almost any combination of pointer followers, array selectors, and record selectors can occur in an access path.

Exercise 5-15: Write an expression that returns the `hrs` field of the departure time of the `plane` recorded pointed to by `p`.

Type Equivalence Was Not Clearly Specified

The Revised Pascal Report states that an expression can be assigned to a variable if the expression and variable have "identical type." Unfortunately, the report does not specify what it means for two types to be identical, and different implementers have interpreted this phrase in different ways. Although the ISO Pascal Standard clears up this ambiguity, it is instructive to consider the possible interpretations.

One such interpretation is called *structural equivalence*, because two types are considered equivalent if they have the same structure. Consider these two variable declarations.

```
var x: record id: integer; weight: real end;
    y: record id: integer; weight: real end;
```

Is the assignment `x := y` legal? The Structural Equivalence Rule says "yes" since the types associated with the two variables have the same structure (i.e., the same description). Next, consider these declarations:

```
type  person = record id: integer; weight: real end;
      car    = record id: integer; weight: real end;
var   x: person;
      y: car;
```

The Structural Equivalence Rule would still allow `x := y` since `person` and `car` are just two different names for what amounts to the same type. Structural equivalence can be defined as follows: Two objects are considered to have the same type if the *structural descriptions* of their types are the same (i.e., word for word).

This last example suggests the problem with structural equivalence: If programmers declare two types, called `person` and `car`, then they probably intend them to represent different things, and it probably does not make any sense to assign one to the other. The fact that they happen to be defined by the same record structure may be a coincidence. This leads to another rule for type equivalence, called *name equivalence.* The Name Equivalence Rule

the mechanism that implements it (e.g., the top pointer of the stack or the array containing its elements). This problem, which is related to indiscriminate access, can severely complicate maintenance.

4. It should be possible to distinguish different types of access. For example, it should be possible to give some users read-only access to a data structure and others read-write access. This helps to solve the side effect and vulnerability problems.

5. Declaration of definition, name access, and allocation should be decoupled. In block-structured languages these functions are usually closely connected. For instance, by declaring a variable in an Algol block (1) the name is defined by its appearance in the declaration, (2) name access is determined by its occurrence in the block since that block and all inner blocks implicitly inherit access to the variable, and (3) storage allocation and deallocation are determined since they will occur simultaneously with entry to and exit from the block. These are really three orthogonal (i.e., independent) functions. In a few cases, languages attempted to decouple these functions. Algol decouples name definition and access from allocation with its **own** variables; Pascal accomplishes the same with its dynamically allocated storage (`new` and `dispose`). Proper separation of these functions would help to solve most of the problems of block structure. We will see later that although Ada has not abandoned block structure, its packages and other related mechanisms eliminate many of the block's shortcomings.

Parnas's Principles

At about the same time that Wulf and Shaw were doing their work, Parnas enunciated two important principles of information hiding. In the introduction to this chapter, we discussed the general idea of information hiding: Each difficult design decision should be hidden inside a module. This rule determines what should be in each module. The two principles we will see next guide us in designing the interfaces between modules.

Parnas's Principles

1. One must provide the intended user with *all* the information needed to use the module correctly *and nothing more.*
2. One must provide the implementor with *all* the information needed to complete the module *and nothing more.*

Thus, the user of a module does not know how it is implemented and cannot write programs that depend on the implementation. This makes the module more maintainable since implementors know exactly what they can and cannot change without impacting the users. Similarly, implementors have no knowledge of the context of use of their module, except that provided in the interface. This simplifies maintenance of programs that use the module because programmers know what they can safely change and what they cannot. We will see that the Ada package construct directly supports Parnas's principles.

Packages Support Information Hiding

The Ada construct that supports the information-hiding principles and controls access to declarations is the *package*. The declaration of a package is broken down into two parts—an interface specification and a body. The interface specification defines the interface between the inside and the outside of the package; hence, it is that information about the package that must be known to the user; and that information about the way it will be used that must be known to the implementor. The package specification is effectively a contract between the user and the implementor of the package. A package specification has the following form:

```
package Complex_Type is
        . . .specification of public names. . .
end Complex_Type;
```

Between the brackets of the package specification (`package-end`), all the specifications of the public names (i.e., the names in the interface) are written. A partial specification of a package that provides complex arithmetic is shown in Figure 7.6.

Figure 7.6 shows that the package `Complex_Type` provides a type (`Complex`), a constant (`I`), and several functions (`Re`, `Im`), and that it overloads the arithmetic operators. The function definitions are specified in the usual way: The types of the parameters and the returned value are specified.

The type `Complex` is also listed in the interface, but it is defined to be a *private* type. This means that although the name `Complex` is visible (and hence may be used in object declarations, parameter specifications, etc.), the internal structure of `Complex` numbers is hidden from users of the package. If this were not done, it would be possible for users to access directly the components of `Complex` numbers without going through the `Re` and `Im` functions. This would interfere with later maintenance if the implementor decided to use a different representation for `Complex` numbers. Notice that there is an appendage to the package specification introduced by the word `private`. This private part of the package includes a definition of the `Complex` type that specifies its representation. What is this information doing in the specification? This is a concession that the Ada designers have been forced to make so that packages will not be too difficult to compile. Users of the `Complex_Type` package will want to declare objects of type `Complex`, which will require the compiler to allocate storage for these records; thus, the compiler must know the representation of `Complex` numbers. This is especially necessary if the program using `Complex_Type` and the package defining it are compiled separately; under these circumstances only the specification of `Complex_Type` is available to the compiler when it is compiling the program using the package.

```
package Complex_Type is
   type Complex is private;
   I: constant Complex;
   function "+"  (X,Y : Complex) return Complex;
   function "-"  (X,Y : Complex) return Complex;
   function "*"  (X,Y : Complex) return Complex;
   function "/"  (X,Y : Complex) return Complex;
   function Re   (X : Complex) return Float;
   function Im   (X : Complex) return Float;
   function "+"  (X : Float; Y : Complex) return Complex;
   function "*"  (X : Float; Y : Complex) return Complex;
private
   type Complex is
      record Re, Im : Float := 0.0; end record;
   I : constant Complex := (0.0, 1.0);
end Complex_Type;
```

Figure 7.6 Specification of Complex Arithmetic Package

We can see that this package also defines a public constant, `I`, defined as a *deferred* constant. Its value must be deferred because it depends on the representation of `Complex` numbers, which is private. The actual definition of the constant is given in the private part of the specification along with the type definition.

The package body, which is known only to the implementor, gives the definition of each name mentioned in the specification. It may also declare any local procedures, functions, types, and so on, needed by this implementation; all of these are private. Part of the implementation of `Complex_Type` is shown in Figure 7.7.

```
package body Complex_Type is
   function "+" (X,Y : Complex) return Complex is
   begin
     return (X.Re + Y.Re, X.Im + Y.Im);
   end;
   function "*" (X,Y : Complex) return Complex is
     RP: constant Float := X.Re*Y.Re - X.Im*Y.Im;
     IP: constant Float := X.Re*Y.Im + X.Im*Y.Re;
   begin
     return (RP,IP);
   end;
   function Re (X : Complex) return Float is
   begin return X.Re; end;
   function Im (X : Complex) return Float is
   begin return X.Im; end;
   function "+" (X : Float; Y : Complex) return Complex is
   begin
     return (X + Y.Re, Y.Im);
   end;
   -- other definitions
end Complex_Type;
```

Figure 7.7 Partial Implementation of a Complex Arithmetic Package

Exercise 7-12: Complete the definition of the package `Complex_Type`.

Exercise 7-13*: We have seen that the private part of a package specification mixes representation information important only to the implementor with the interface information needed by the user. Describe an alternative that does not mix things up this way but still allows a compiler to allocate storage for objects of private types. You may alter Ada's package declarations or describe an alternative method for the compiler to get the needed information.

Name Access Is by Mutual Consent

We have seen that the implementor of a package can control, by the placement of the declarations in the public part or the private part of the package, which names can be accessed by a user of the package. Anything placed in the specification is public and potentially accessible. A user gains access to the publics of a package with a `use` declaration, as shown:

```
declare
  use Complex_Type;
  X,Y : Complex;
  Z : Complex := 1.5 + 2.5*I;
begin
  X := 2.5 + 3.5*I;
  Y := X + Z;
  Z := Re(Z) + Im(X)*I;
  if X = Y then X := Y + Z;
  else X := Y*Z; end if;
end;
```

The `use` declaration makes all of the public names of the package visible throughout the block in which it appears. We can see that this permits using all of the types (`Complex`), functions (`Re`, `+`), constants (`I`), and so forth, as though they were built in. (In fact, the Ada language is defined as though the "built-in" types are defined in packages that are automatically used for all programmers.) Thus, name access is by mutual consent: The package implementor determines which attributes are to be public and the package user decides whether to *import* the attributes of a particular package. In fact, Ada provides even more control to package users since, if they do not need all of the names defined by a package, they can select just the ones they want. This is done with a dot notation similar to Pascal's (e.g., `Complex_Type.I`) or by a variant of `use` that we will not discuss here.

A compilation comprises one or more "library items," which are often packages. Whether they are part of the same compilation or not, they are not mutually visible without explicit declaration. For one item to be visible to another, the latter must have a *context clause* mentioning the former. For example, a module needing to use complex numbers should be preceded by `with Complex_Type`. (Then the names can be accessed by the dot notation, e.g., `Complex_Type.Re`; they will be directly visible, without using the dot, if the `with` is followed by `use Complex_Type`.)

Packages as Libraries

We have just seen how to use a package to define an *abstract data type*. There are also many other ways that packages can be used to modularize programs; some of these are discussed in the following sections. One of the simplest, which is really a degenerate form of an abstract data type, is a *library*. Suppose we wished to define a `Plot` library that provided subprograms for plotting. This can easily be specified as a package:

```
package Plot is
  type Point is record X,Y : Float; end record;
  procedure Move_To (Location : Point);
  procedure Line (From, To : Point);
  procedure Circle (Center : Point; Radius : Float);
  procedure Fit (Data : array (Integer range < >) of Point);
  function Where return Point;
end Plot;
```

Then, if users wish to do some plotting in a module, they only have to include a `with Plot` request in the context clause preceding that module. Of course, the private part of the package may include the definitions of constants and subprograms that are needed by the implementation but are hidden from users. You can see that a library is just a package that does not contain any data structures.

Packages Permit Shared Data Areas

We have just looked at packages that contain procedures but no data structures; we will now look at the opposite—packages that contain data structures but no procedures. For example, suppose we wanted a buffer to be used for communicating characters between two subprograms. This could be done by the declaration

```
package Communication is
  In_Ptr, Out_Ptr : Integer range 0..99 := 0;
  Buffer : array (0..99) of Character := (0..99 => ' ');
end Communication;
```

(The declaration of `Buffer` makes it an array initialized to all blanks.) Given this definition of `Communication`, two procedures `P` and `Q` can use it for communication by including a `use` for the package:

```
with Communication; use Communication;
procedure P is
  use Communication;
begin
     :
  Buffer(In_Ptr) := Next;
```

```
   In_Ptr := (In_Ptr + 1) mod 100;
       :
end P;
with Communication; use Communication;
procedure Q is
   use Communication;
begin
       :
   C := Buffer(Out_Ptr);
       :
end Q;
```

This way of using packages is similar to the way labeled COMMON is used in FORTRAN. It also solves the problem of overlapping definitions discussed in the section on encapsulation; only those subprograms that need access to `Buffer` will have `with Communication; use Communication`.

Packages Can Be Data Structure Managers

When we first saw the idea of information hiding, we said that each module should encapsulate one difficult design decision; we also said that the choice of the representation for a data structure was often such a difficult decision. Therefore, a common use of Ada packages is to encapsulate a data structure and provide a representation independent interface for accessing it.

We can take a stack data structure as a common case. When we design a data structure, one of the first questions we must ask is: "What operations are to be available on this data structure?" We will immediately come up with `Push` and `Pop`; there are others, however. What will happen if we try to `Pop` an element from an empty stack? Surely we will generate an error, but it is preferable for the users to have an `Empty` test so that, if they are unsure about whether the stack is empty, they can test it before they do a `Pop`. We may also want a `Full` test to determine if there is any room in the stack before we do a `Push`. Finally, we will need an error signal or *exception*, `Stack_Error`, which is raised if someone does a `Pop` from an empty stack, and so forth. We also need to know the sort of things that the stack can hold; we will assume they are integers. We now have the information necessary to specify the `Stack1` package:

```
package Stack1 is
  procedure Push (X : in Integer);
  procedure Pop (X : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
  Stack_Error : exception;
end Stack1;
```


The difficult design decision—whether to represent the stack as an array, linked list, or something else—is hidden in the package.

Once the package has been implemented, and made accessible if necessary by `with`, it can be used as before. For example, if we intend to use the stack over a large part of the program, we can make its names available with a `use`:

```
declare
  use Stack1;
  I, N : Integer;
begin
    :
  Push(I);
  Pop(N);
    :
  if Empty then Push(N); end if;
    :
end;
```

We can also use the "dot" notation to select a public attribute from `Stack1` without using the `use`, for example,

```
Stack1.Push(I);
Stack1.Pop(N);
if Stack1.Empty then Stack1.Push(N); end if;
```

This allows users of the package to be as selective as necessary about the names imported from `Stack1`; again, name access is by mutual consent.

How would we go about implementing the stack package? For the sake of this example, we will assume that we have decided on an array representation for the stack. The implementation of the stack package is shown in Figure 7.8. We can see that this implementation of `Stack1` has two private names: `ST`, the array that holds the stack elements, and `Top`, the pointer to the top of the stack. These are completely invisible and inaccessible to users of the stack; any attempt to access them, for example, by `Stack.ST` or `Stack.Top`, will be diagnosed by the compiler as a program error. The `raise Stack_Error` statement is an example of an *exception*. Exceptions are discussed later (Chapter 8).

Exercise 7-14*: Write a package body that implements `Stack1` using linked lists. To do this you will need to know a few details about Ada: (1) If `T` is a type, then `access T` is the type of pointers to things of type `T`. (2) If `T` is a record type, then `new T(X1, ..., Xn)` allocates an instance of that record type and returns a pointer to that record. What is the meaning of the `Full` function in a linked implementation of stacks?

```
package body Stack1 is
   ST : array (1..100) of Integer;
   Top : Integer range 0..100 := 0;
   procedure Push (X : in Integer) is
   begin
     if Full then raise Stack_Error;
     else
       Top := Top + 1; ST(Top) := X;
     end if;
   end Push;
   procedure Pop (X : out Integer) is
   begin
     if Empty then raise Stack_Error;
     else
       X := ST(Top);
       Top := Top - 1;
     end if;
   end Pop;
   function Empty return Boolean is
   begin return Top = 0; end;
   function Full return Boolean is
   begin return Top = 100; end;
end Stack1;
```

Figure 7.8 Body of Simple Stack Package

Generic Packages Allow Multiple Instantiation

Suppose that the program we are writing requires two stacks. To get a second stack, we will have to repeat the entire definition of `Stack1` with only its name changed:

```
package Stack2 is
  procedure Push (X : in Integer);
  procedure Pop (X : out Integer);
  function Full return Boolean;
  function Empty return Boolean;
  Stack_Error : exception;
end Stack2;
package body Stack2 is
  ... all of the definitions exactly as they
  ... appeared in Stack1.
end Stack2;
```


It is clearly a waste of time to have to copy the entire definition of `Stack1` verbatim. Even if this copying is done automatically, say with an editor, it will still create a maintenance problem. Whenever a bug is corrected or the implementation of the package is changed, the modification will have to be repeated for each copy; there is a much greater chance of error. This approach is also inferior from the standpoint of readability since it is not obvious to someone trying to understand the program that the two stacks are really the same; they will have to compare the definitions line by line to determine this. Clearly, what we have here is a failure to modularize; the separate copies of the package should be abstracted out so that they have to be written and maintained only once, which is an example of the Abstraction Principle. This ability is provided by Ada's *generic* facility.

We can see the motivation for this facility by looking at the way that programming languages have solved similar abstraction problems. When we need to repeat the same control sequence several times with different data, we define a procedure that implements the control sequence and then call it with the required data. When we need to repeat the same data structure several times in different storage areas, we define a data type that specifies a *template*, or pattern, for the data structure, and then we use that type in variable declarations so as to create multiple *instances* of the data structure. This is exactly the approach taken with packages. A template for packages, called a *generic package*, is defined. The template can be used to repeat the package by *generic instantiations*. Let's see how this works. A template for a generic stack package would be written

```
generic
package Stack is
  procedure Push (X : in Integer);
  procedure Pop (X : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
  Stack_Error : exception;
end Stack;
```

We can see that this looks exactly like our previous specification of the stack package except that the word `generic` has been appended to its front. This is what informs us that we are defining a template for stacks and not a particular stack. The body for the generic stack package is exactly like that in Figure 7.8 so we will not repeat it.

We have seen how to write a template for a generic package. Next we must investigate the instantiation of these templates. Suppose we want two stacks called `Stack1` and `Stack2`. We can request the creation of two instances, or copies, of the template `Stack` with the generic instantiations:

```
package Stack1 is new Stack;
package Stack2 is new Stack;
```

These create two copies of the data areas defined by `Stack`, which are associated with the names `Stack1` and `Stack2`; the procedural code (for `Push`, `Pop`, etc.) can be shared by the instances. The two instances of `Stack` can be used with the dot notation, for example,

```
Stack1.Push(I);
Stack2.Pop(N);
if Stack1.Empty then Stack1.Push(N); end if;
```

Notice that it is not possible to use the `use` construct to enter both stacks into the same scope since procedure calls like `Push(I)` would be ambiguous; there would be no way to tell to which stack the `Push` referred.

You have probably already noticed that package instantiation is analogous to the instantiation of procedures, which we discussed in Chapter 3 on Algol-60. In that case, we create a new activation record, or instance, for a procedure, which contains all of its local variable storage but shares the executable code with other instances. When we study object-oriented languages in a later chapter, we will see that these ideas are very closely related. One difference that must be pointed out here is that while procedures may be *dynamically instantiated*, Ada allows only the *static instantiation* of packages; that is, each instance is associated with a declaration and the number of declarations is determined by the structure of the program. (Note, however, that a package declaration may be a local to a procedure that is dynamically instantiated; thus, there can be one instance of the package for each instance of the procedure. This is the only sense in which Ada packages can be dynamically instantiated.) Some other languages, for example, Simula-67 and Smalltalk (Chapter 12), allow the dynamic instantiation of packages in much the same way that records can be dynamically allocated.

■ ***Exercise 7-15*:*** Discuss the dynamic instantiation of packages. Is there any need for this facility? Why do you suppose it was left out of Ada? Discuss how such a facility might be included in Ada, and any other mechanisms that would have to be included to support it. Are there any efficiency consequences to dynamic instantiation? How about simplicity or readability consequences?

Instances May Be Parametrically Related

The generic facility has more capabilities than the simple copying of templates. In the generic stack package we have seen defined, the stack was limited to a size of 100. Now suppose that instead of two equal-size stacks we needed `Stack1` to be of size 100 and `Stack2` to be of size 64. How would we accomplish this? It may seem that we would have to recopy the definition of `Stack` with all of the occurrences of `100` replaced by `64`. Again, this would be a very inefficient thing to do; it would hurt writability, readability, and maintainability. The analogous problem with procedures is solved by parameters; parameters allow the data to vary from one procedure call to another. Ada adapts this approach to packages by allowing parameters on a generic specification. For example, to allow the length of stack to vary from instance to instance, the package is specified.

```
generic
  Length : Natural := 100;
package Stack is
  procedure Push (X : in Integer);
  procedure Pop (X : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
  Stack_Error : exception;
end Stack;
```


Notice that the `Length` parameter has been given a default value of 100; this is the value it will have if it is not specified. The body of the package is altered by replacing each occurrence of 100 by `Length` as shown in part here:

```
package body Stack is
  ST : array (1..Length) of Integer;
  Top : Integer range 0..Length := 0;
  ... the rest of the definitions,
  ... but with 100 replaced by Length
end Stack;
```

When a stack is instantiated, this parameter must (if not omitted) be bound to a natural number. We can get the 100- and 64-element stacks by

```
package Stack1 is new Stack(100);
package Stack2 is new Stack(64);
```

Since the default stack length is 100, the first instantiation could have been written

```
package Stack1 is new Stack;
```

Generic packages can have any number of parameters of any types, just as procedures do. They can also have several types of parameters that procedures cannot have. Suppose that we needed to use a stack, `Stack3`, that contains only characters. Again it would seem that we must copy the entire definition of `Stack` with every occurrence of `Integer` replaced by `Character`. This is undesirable for all of the reasons we have already discussed; instead it is handled by generics. A specification of type-independent stacks is

```
generic
  Length : Natural := 100;
  type Element is private;
package Stack is
  procedure Push (X : in Element);
  procedure Pop (X : out Element);
  function Empty return Boolean;
  function Full return Boolean;
  Stack_Error : exception;
end Stack;
```

The type parameter is defined to be *private* because it acts like a private type: The only operations available on it (within the package) are assignment and equality comparisons. There are other forms of type parameters in generic packages that allow more operations but on a restricted class of objects; this is too detailed to warrant our attention here. The implementation of `Stack` is shown in Figure 7.9. Given these definitions, the instantiation of general stacks is accomplished as before, for example,

```
package Stack1 is new Stack (100, Integer);
package Stack3 is new Stack (256, Character);
```

These stacks can be used with the dot notation as before, for example, `Stack1.Pop(N)` or `Stack3.Push('A')`. They can also be referred to without the dot notation through the `use` construct:

```
declare
   use Stack1;
   use Stack3;
   I, N : Integer;
   C, D : Character;
begin
   Push(I);
   Push(C);
   Pop(N);
   Pop(D);
   if Stack1.Empty then ...
   if Stack3.Full then ...
end;
```

"Using" both stacks is permitted because context determines which procedure is intended. For example, `Push(I)` is unambiguous because `I` is an `Integer` and there is only one `Integer Push` procedure visible (the other `Push` procedure works on `Characters`). In such a situation, the `Push` procedure is said to be *overloaded* since it bears several meanings at once. This is analogous to the overloaded enumeration-type elements previously discussed and to the built-in overloaded operators (e.g., '+' works on `Integers`, `Floats`, and `Complexes`). Notice that context cannot be used for the `Empty` and `Full` functions because they do not have any arguments (or return values) that depend on the element type; these functions must still be accessed with the dot notation.

Generic Packages Are Difficult to Compile

All of these convenient facilities are not without cost; efficient generation of code for generic packages can be very complicated. We consider a few of the issues in this section. We saw earlier that generic packages without parameters could be instantiated much as procedures are instantiated: A new data area is created for each instance and the executable code is shared by all of the instances. This case is illustrated by Figure 7.10. This same kind of sharing is possible with most simple kinds of parameterization; for example, the generic `Stack` package parameterized just by `Length` can make use of shared code if the differing information, the array length, is stored with the instance. It is also necessary to use the most gen-