3 GENERALITY AND HIERARCHY: ALGOL-60

3.1 HISTORY AND MOTIVATION

An International Language Was Needed

Already in the mid-1950s it was becoming apparent to some computer scientists that a single, universal, machine-independent programming language would be valuable. The problem, of course, was portability; since each machine had its own instruction sets, assemblers, and pseudo-code interpreters, it was very difficult to transport programs from one machine to another. As early as 1955,[1] GAMM, a European association for applied mathematics and mechanics, had set up a committee to study the development of such a language. The problem was serious enough that at a conference in Los Angeles in May 1957, representatives of ACM (The Association for Computing Machinery) and three manufacturer's users' groups suggested that ACM form a committee to study and recommend action for the creation of a universal programming language. In June ACM appointed representatives of the computer industry, users, universities, and the federal government to such a committee. In October GAMM proposed to ACM that a joint effort be undertaken, to which ACM agreed. In this proposal GAMM made several points; for example, that no existing language was so popular that it should be chosen as the standard and that another "nonideal" language would not solve the problem of the proliferation of languages. GAMM also stressed that the passing of time aggravated the problem since each month brought more programming languages and more programs written in those languages. A joint meeting of four members of each of the ACM and GAMM subcommittees was scheduled for May and June 1958 in Zurich.

The Algol-58 Language Was Developed

The language now known as Algol-58 was designed by the eight representatives who met in Zurich. The European members brought several years of work on algebraic language design

[1] Most of the historical information in this section comes from the "History of Programming Languages Conference Proceedings." *SIGPLAN Notices 13*, 8 (August 1978).

and the Americans brought their experience in implementing pseudo-codes and other programming systems. For example, John Backus was one member of the American group. This combination of talents proved very productive, and in *eight days* the language was essentially completed. In December 1958 Perlis and Samelson, writing for the committee, published the "Preliminary Report—International Algebraic Language," in the *Communications of the ACM*. At that time the official name of the new language was IAL, although the acronym Algol (Algorithmic Language) had already been proposed. It is for this reason that the 1958 document is often known as the "Algol-58 Report."

It is instructive to see the objectives of the new language as stated in the Algol-58 Report:

I. The new language should be as close as possible to standard mathematical notation and be readable with little further explanation.
II. It should be possible to use it for the description for computing processes in publications.
III. The new language should be mechanically translatable into machine programs.

As our description in this book progresses, you should try to decide how well Algol met these goals.

Algol-58 created a great stir when it was announced; implementations were begun at many universities and laboratories. Indeed, many of IBM's users even suggested that IBM abandon FORTRAN and throw all of its support behind Algol; IBM, however, decided against that course. Many dialects of Algol-58 appeared, including NELIAC, the Navy Electronic Laboratories International Algol Compiler, and JOVIAL, which was widely used by the Air Force. JOVIAL is an acronym for "Jules' Own Version of the International Algebraic Language," reflecting the alterations made to IAL (Algol-58) by Jules Schwartz. By jumping too soon on the Algol bandwagon, these efforts diminished Algol's value as a universal language since they committed their users to an obsolete version of Algol.

A Formal Grammer Was Used for Syntactic Description

The Algol-58 Report was only a preliminary specification; it was intended that critiques and suggested improvements would be collected until November 1959, when the final language would be designed. One of the media for exchange of Algol information was the International Conference on Information Processing held by UNESCO in Paris in June 1959. At this conference John Backus presented a description of Algol using a formal syntactic notation he had developed. Peter Naur, then the editor of the *Algol Bulletin*, was surprised because Backus's definition of Algol-58 did not agree with his interpretation of the Algol-58 Report. He took this as an indication that a more precise method of describing syntax was required and prepared some samples of a variant of the Backus notation. As a result, this notation was adopted for the Algol-60 Report and is now known as BNF, Backus–Naur Form, reflecting the contributions of both men. This important method for describing programming languages is discussed in the next chapter.

Algol-60 Was Designed

Thirteen members of ACM and GAMM met in Paris for six days in January 1960 in order to prepare a final report on the language incorporating the various suggestions that had been received. The language that resulted, Algol-60, was very different from Algol-58 and was described in a report published in May 1960. The resulting language was remarkable for its generality and elegance, particularly considering that it was designed by a committee. Alan Perlis has described it as "more of a racehorse than a camel." A few remaining errors and ambiguities were corrected at a meeting in Rome in 1962, and the "Revised Report on the Algorithmic Language ALGOL-60" was published in the *Communications* in January 1963. Algol continued to evolve, a process that we discuss at the end of this chapter. We will concentrate on Algol-60, since it illustrates best the characteristics of a second-generation language.

The Report Is a Paradigm of Brevity and Clarity

At a time when programming language descriptions often stretch to hundreds or thousands of pages, it is remarkable to realize that the original Algol-60 Report was 15 pages long. The brevity and clarity of this report contributed significantly to Algol's reputation as a simple, elegant language. How was it possible to produce so short a report? One reason was the use of the BNF notation; this provided a simple, concise, easy-to-read, precise method of describing Algol's syntax. We discuss it in the next chapter.

BNF is useful for describing only the *syntax* of a language. How was the *semantics*, or meaning, of Algol's constructs described? The committee decided to use clear, precise, unambiguous English-language descriptions, which resulted in a report that was readable by potential users, implementers, and language designers. The clarity and brevity of the Algol-60 Report have rarely been achieved since; it remains a standard against which all programming language descriptions can be compared.

3.2 DESIGN: STRUCTURAL ORGANIZATION

Figure 3.1 displays a small Algol-60 program to compute the mean (average) of the absolute value of an array. In Algol, the reserved words of the language (e.g., '**if**') are distinct from the identifiers (e.g., 'if'), therefore, it is common to print the reserved words in boldface or to underline them, as can be seen in this example. These lexical conventions are discussed in Chapter 4, Section 4.1.

Algol Programs Are Hierarchically Structured

One of the primary characteristics and important contributions of Algol is its use of *hierarchical structure*, or nesting, throughout its design. For example, an Algol program is composed of a number of nested environments, as we can see in the contour diagram in Figure 3.2, which corresponds to the program in Figure 3.1. The nesting of environments is discussed in Section 3.3, on name structures.

Algol-60 also allows control structures to be nested. For instance, a **for**-loop, such as

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

```
begin
  integer N;
  Read Int (N);

  begin
    real array Data[1:N];
    real sum, avg;
    integer i;
    sum := 0;

    for i := 1 step 1 until N do
      begin real val;
        Read Real (val);
        Data[i] := if val < 0 then-val else val
      end;
    for i := 1 step 1 until N do
      sum := sum + Data[i];
    avg := sum/N;
    Print Real (avg)
  end
end
```

Figure 3.1 An Algol-60 Program

can be made the object of an **if**-statement. For example,

```
if N > 0 then
  for i := 1 step 1 until N do
    sum := sum + Data[i]
```

This nesting greatly decreases the number of **goto**-statements required in a program. Some of its implications are discussed in Section 3.5 on control structures.

Constructs Are Either Declarative or Imperative

As in FORTRAN, the constructs of Algol-60 can be divided into two categories—*declarative* and *imperative*. The declarative constructs bind names to objects (variables and procedures in Algol's case) and the imperatives do the work of computation.

There are three kinds of declarations in Algol-60—variable declarations, procedure declarations, and **switch**-declarations. Variable declarations are similar to FORTRAN's, except that the only data types allowed are **integer, real,** and **Boolean;** for example,

```
integer i, j, k
```

The array declarations are analogous, although the lower bound is allowed to be numbers other than 1:

```
real array Data[-50:50]
```

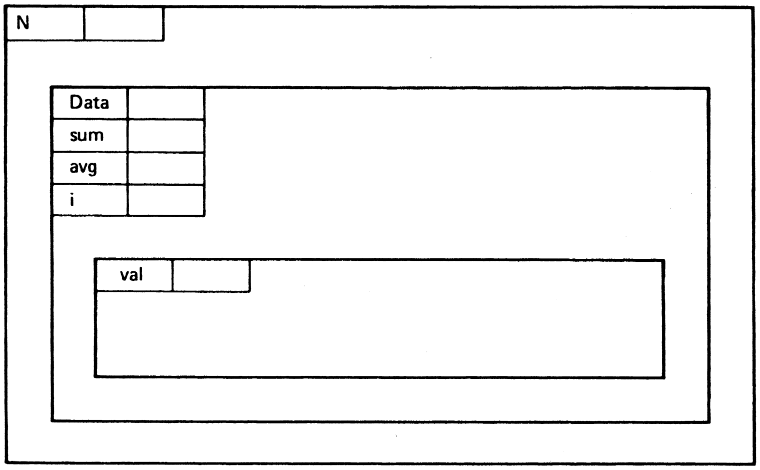

Figure 3.2 Contour Diagram of an Algol Program

Algol has *dynamic arrays*, that is, their bounds can be computed at run-time; this will be discussed in Section 3.4 on data structures.

Algol uses the term *procedure* to refer to a subprogram and distinguishes between *typed* procedures (Algol's name for a *function*) and *untyped* procedures, which are like FORTRAN subroutines. Since there are no implicit declarations in Algol, procedure declarations are required to specify the types of their formal parameters; for example,

```
real procedure dist (x1, y1, x2, y2);
  real x1, y1, x2, y2;
  dist := sqrt ((x1-x2)↑2 + (y1-y2)↑2)
```

In Section 3.5 on control structures, we will take an in-depth look at procedures.

Finally, the **switch**-*declaration* serves the same function as the FORTRAN computed GOTO, namely, breaking a problem down into cases; this is discussed in Section 3.5 on control structures.

Imperatives Are Computational and Control-Flow

There are two classes of imperative constructs in Algol-60—the computational and the control-flow. There are no input-output constructs in Algol-60; it was intended that input-output be handled by library procedures. As in FORTRAN the only computational statement is the *assignment*, which has the form 'variable := expression', where 'variable' is a simple variable or an array reference and 'expression' is an integer, real, or Boolean expression constructed from variables, constants, and operators. Algol provides the standard arithmetic operators (+, −, etc.), relational operators, which return a Boolean result (=, >, ≤, etc.), and Boolean operators (∧, ∨, ¬, etc.). All of these are organized by precedence to give a natural appearance to the notation.

The assignment operation ':=', deserves some comment. FORTRAN had used an equal sign for assignments by the analogy with mathematical definition, for example, 'let *C* =

$2\pi r$'. It was widely recognized that this notation was really not very accurate, since assignment and definition are really two different things. For example, `I = I + 1` makes no sense as a definition, although it is a very useful assignment operation. Early programming notations (and indeed other engineering notations) commonly used a rightward pointing arrow to indicate assignment. For instance, `I + 1` $\rightarrow$ `I` indicates that the value of `I + 1` is to be put into `I`. A number of notations had been used for the assignment arrow in early programming languages, including $=>$ and $\geqslant$, but the limitations of input devices led the Algol designers to compromise on '=:' as an arrow symbol. Later, on the basis of experience with FORTRAN, this was turned around into a leftward assignment: `i := i + 1`. Almost all programming languages now use this notation.

A significant exception is C and its derivatives, which return to FORTRAN's '=' for assignment and invents '==' for equality. Unfortunately, C's weak typing permits '=' to be written where '==' is intended, which can cause obscure program bugs. This is one of C's many second-generation characteristics. (See Section 5.6 for more on C.)

Algol Has the Familiar Control Structures

All other Algol imperatives alter the flow of control in the program. The most obvious of these is the **goto**-statement, which transfers to a labeled statement. Algol has a conditional statement, the **if-then-else** and an iterative statement, the **for**-loop which is an elaboration of FORTRAN's `DO`-loop. Finally, it has a procedure invocation, as we have seen. Control structures are the subject of Section 3.5.

The Compile-Time, Run-Time Distinction Is Important

Algol-60 programs typically go through a compilation process very much like that of FORTRAN programs. There are a few important differences, however. FORTRAN is completely static, that is, all of the data areas are allocated and arranged by the compiler. By the time the program is loaded into memory, it is ready to run, with only the contents of the memory locations being altered by the program. We will see in Sections 3.3–3.5 of this chapter that various features in Algol (e.g., dynamic arrays and recursive procedures) preclude a static, compile-time layout of memory. Rather, various data areas are allocated and deallocated at run-time by the program. In other words, Algol data structures have a later *binding time* than FORTRAN data structures. More specifically, the name is bound to a memory location at run-time rather than compile-time. As in FORTRAN, it is bound to its type at compile-time.

The Stack Is the Central Run-Time Data Structure

There are many disciplines for organizing the dynamic allocation and deallocation of memory. The one used by Algol (and most other programming languages) is the *stack*. We will see in Sections 3.3–3.5 that Algol programs have one stack that they use for holding activation records for procedures and blocks. Dynamic allocation and deallocation are achieved by pushing and popping these activation records on the stack. Nonstack-oriented run-time structures are discussed in Chapter 12 (Section 12.5).

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

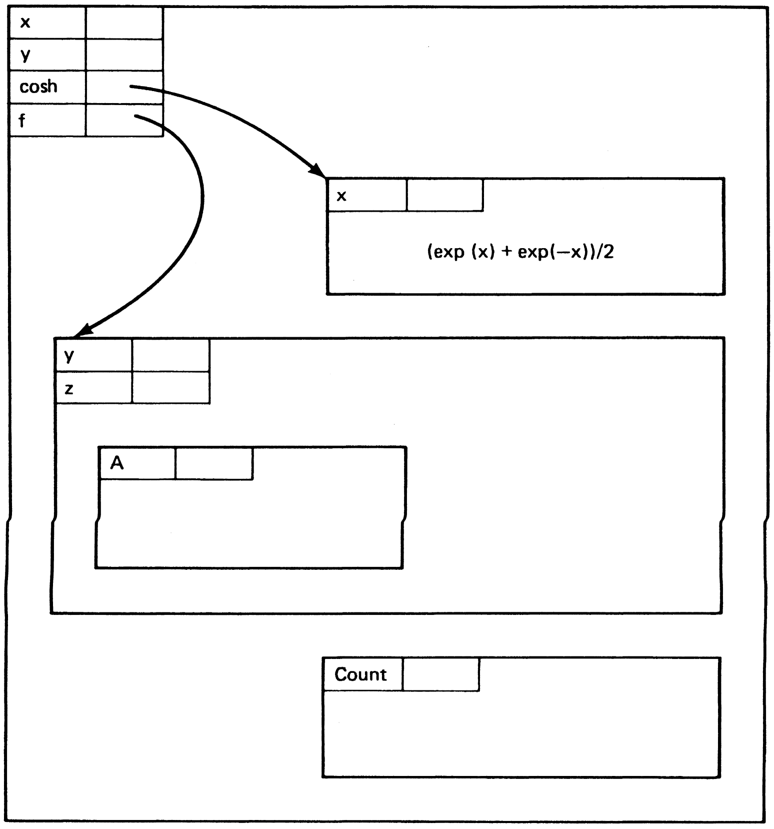

Figure 3.4 Contour Diagram of Nested Scopes

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.

Figure 3.5 Algol Program for Exercise

```
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
```

■ ***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

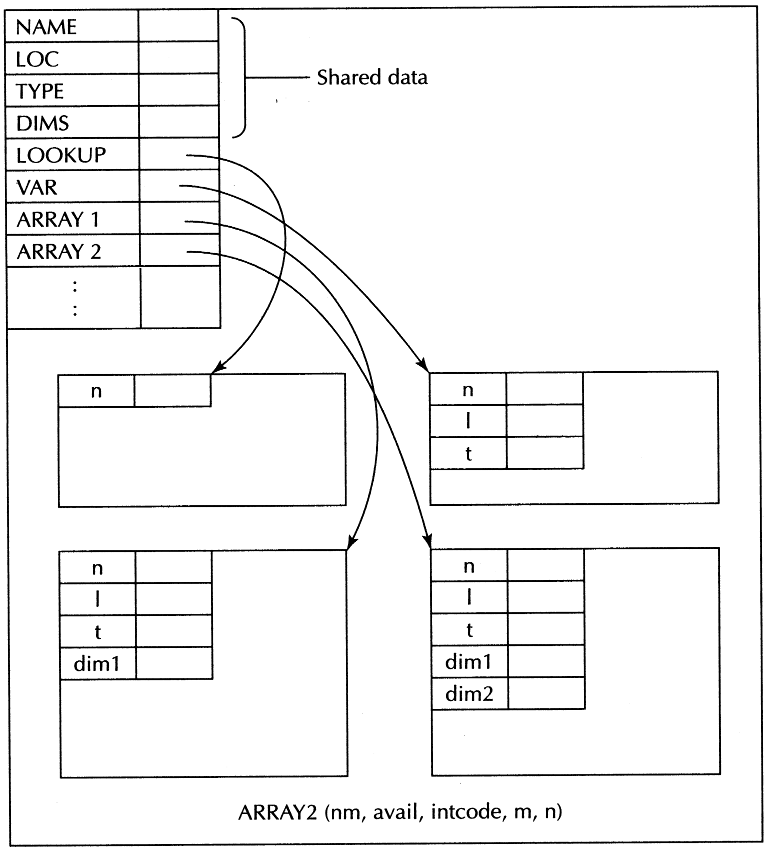

Figure 3.7 Contours Showing Shared Data

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

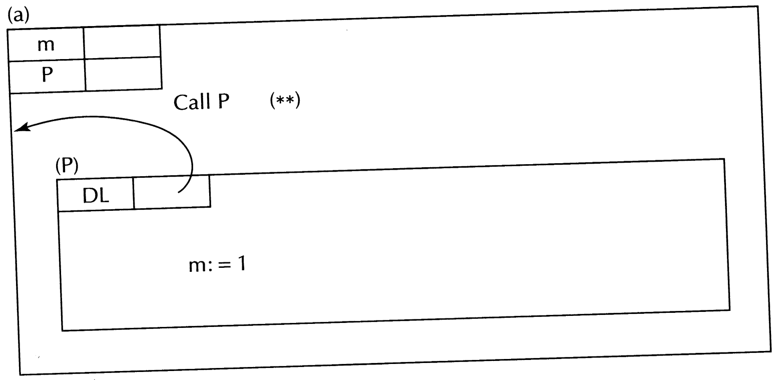

Figure 3.8 Invocation of P from Outer Block (a)

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.

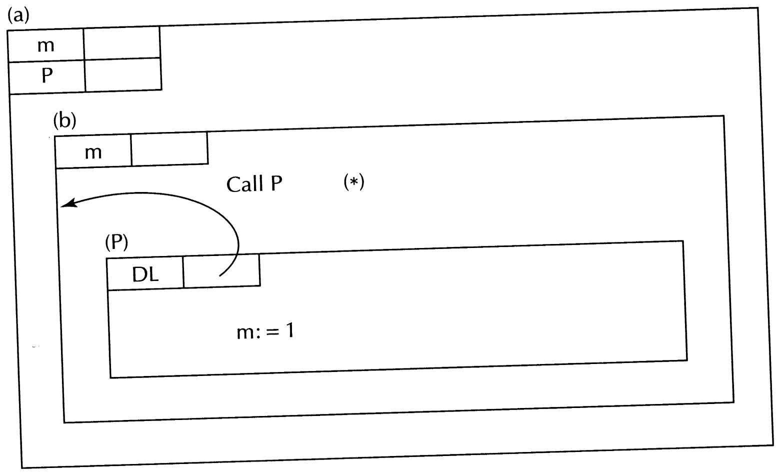

Figure 3.9 Invocation of P from Inner Block (b)

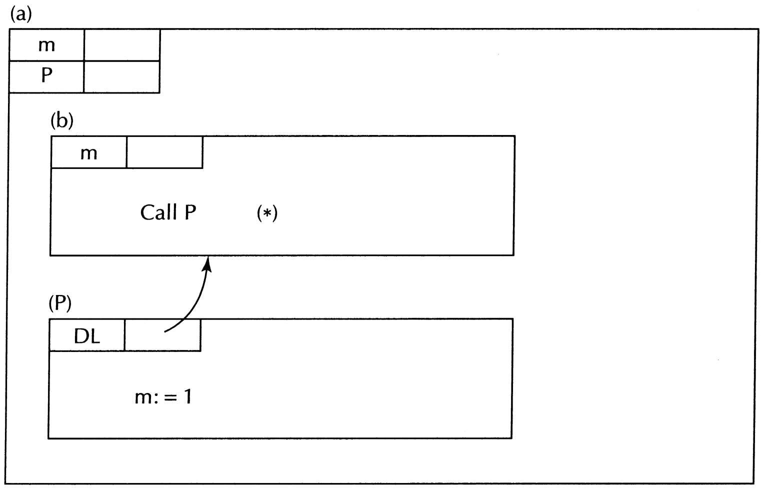

Figure 3.10 Invocation of P when called in Environment of Definition

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 someone uses the same word in the same conversation in two different ways, if the context is switched without warning. This practice can be so detrimental to clear thought that it is classified as a logical fallacy, *equivocation.* The same holds in programs. Programmers will be more likely to write reliable programs if, when they write a statement or expression, they know what it means. A scoping discipline that allows the meaning of procedures to shift and slide depending on their context of use is not conducive to reliable programming. This is the reason that Algol and almost all new programming languages (including newer dialects of traditionally dynamically scoped languages such as LISP) have adopted static scoping.

In summary, we have seen that static scoping causes the static structure of a program to agree more closely with its dynamic behavior than does dynamic scoping. But this is just the Structure Principle, which states that the dynamic behavior of a program should correspond in a simple way to its static structure. Therefore, we can say that static scoping is in accord with the Structure Principle, but dynamic scoping is not.

▩ ***Exercise 3-10:*** Discuss static and dynamic scoping. Do you agree that static scoping is better? Can you think of any ways in which dynamic scoping could be improved without losing its good points?

Blocks Permit Efficient Storage Management on a Stack

In Chapter 2 we saw that the FORTRAN `EQUIVALENCE`-statement was intended to permit a programmer to conserve memory by using it for multiple purposes. We also discussed some of the pitfalls in this mechanism and hinted that newer languages provide a better solution. To understand this better solution, consider this Algol program outline:

```
a:begin integer m, n;
     b:begin real array X[1:100]; real y;
          .
          .
          .
       end
     .
     .
     .
     c:begin integer k; integer array M[0:50];
          .
          .
          .
       end
 end
```

The contour diagram of the above program outline is shown in Figure 3.11. We can see that the two blocks labeled (b) and (c) are *disjoint*, that is, neither is nested in the other. What are the consequences of this? Whenever the program is executing in (b), the local variables of (c), namely `k` and `M`, are not visible since the site of execution is not in their scope. Conversely, whenever the program is executing in (c), the local variables of (b) are not visible. To put it another way, the variables `X` and `y` are never visible at the same time as the variables `k` and `M` because the two sets of variables have disjoint scopes.

It is necessary to make only one assumption in order to turn this fact about disjoint scopes into a solution of the storage-sharing problem. This assumption is that the value of a

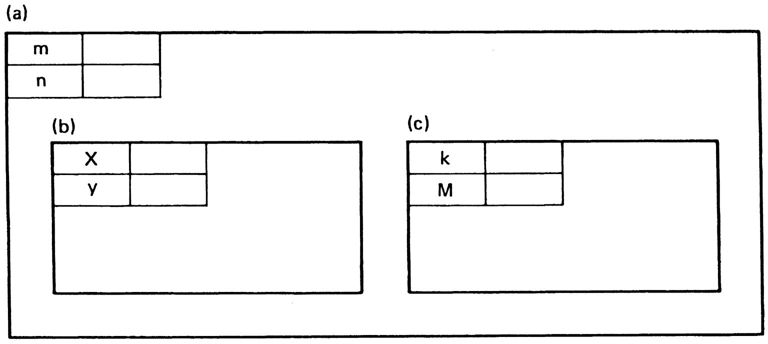

Figure 3.11 Contours of Disjoint Scopes

variable is retained only so long as the program is executing in the scope of that variable or in a block or procedure that will eventually return to the scope of that variable. In other words, if the flow of control leaves the scope of a variable (i.e., leaves the block in which the variable is declared), then the contents of that variable are discarded.[2] That is, when the flow of control enters a block, all of the variables declared in that block become visible but with undefined content—that is, they are uninitialized.

Why does this solve the shared array problem? Consider the previous example (Figure 3-11): The array `X` exists only when the program is executing in block (b) or a block or procedure that can return to (b). Whenever the program leaves this block, the array becomes invisible and its contents are discarded. In other words, for all intents and purposes, when the program is not in block (b), the array `X` does not exist. The same applies to the array `M`; it exists only when the program is within, or can return to, the (c) contour. Since these contours are disjoint, the program can never be in both of them at the same time so the arrays `X` and `M` never exist at the same time. This is the solution to our problem: Since these two arrays can never coexist, they can occupy the same storage locations when they do exist.

Notice that this is much more *secure* than FORTRAN `EQUIVALENCE`. Recall that with `EQUIVALENCE` there was the danger that a program might store into one array (say `M`) while the other (say `X`) was still needed. In Algol this can never happen since the arrays are never visible at the same time. By sharing storage only between disjoint environments, Algol prevents arrays from being corrupted and ensures (at least in this situation) the security of the system.

We have seen that Algol block structure *permits* memory to be used for multiple purposes, but we have not discussed how this is *implemented.* Notice that Algol blocks obey a last-in, first-out discipline. That is, the block that was last entered is the first to be exited, and the block that was first entered (i.e., the outermost) is the last to be exited. This is a simple consequence of the fact that blocks are nested, that is, they are structured hierarchically. Whenever we encounter a last-in, first-out discipline, the data structure that should come to

[2] For situations where this is not desirable, that is, where the value of a variable must be retained from one activation of a block to its next, Algol provides a mechanism called an **own** variable. We will not discuss this further in this book.

mind is a *stack.* In fact, stacks are often called LIFOs (pronounced "lie-foe"), an acronym for "last-in, first-out." Stacks are used for storage allocation in the following way. Whenever a block is entered, an *activation record* for that block is pushed onto the top of the run-time stack. This activation record contains space for all the variables local to that block. Conversely, whenever the block is exited, its activation record will be popped off of the stack, thus freeing its storage for use by other blocks. This process is pictured in Figure 3.12. We can see that the arrays X and M share the same memory locations; storage for them is *dynamically* (i.e., at run-time) allocated and deallocated, and the block structure of Algol ensures that this is done in a secure way *transparent* to the programmer (recall Section 1.4). Before we leave this subject, we must mention that it was not an accident that the storage area associated with a block was called an *activation record*; we will see in Section 3.5 that this use of the term is completely consistent with its use in Chapter 2 to denote the state of a subprogram.

Responsible Design Entails Understanding Users' Problems

The distinction between FORTRAN EQUIVALENCE and Algol blocks illustrates an important issue in programming language design. We can imagine the Algol designers asking FORTRAN programmers about the features they wanted in the new international language, and we can imagine the programmers saying that they would like something like EQUIVALENCE so that they can share memory among arrays. The Algol designers could have decided to satisfy this request by including something like an **equivalence** in Algol, but they did not. Rather, they understood the programmers' problem—the need to share storage among arrays—and provided a better, more secure solution: block-structured storage allocation. This is an example of responsible language design, because the Algol committee applied their language design expertise to the solution of the problem, rather than irresponsibly providing whatever the programmers desired. As Niklaus Wirth, the designer of Pascal, has said, "The language designer must not ask 'what do you want?', but rather 'how does your problem

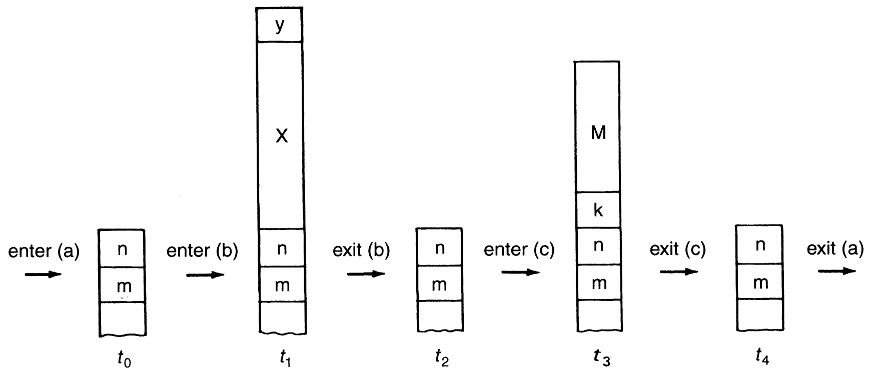

Figure 3.12 Storage Reallocation on a Stack

arise?' " Language designers, not users, should be responsible for the features in languages. We can summarize these considerations in

The Responsible Design Principle

Do not ask users what they want; find out what they need.

3.4 DESIGN: DATA STRUCTURES

The Primitives Are Mathematical Scalars

Like FORTRAN, Algol-60 was intended to be used for scientific applications. Therefore, the primitive data types are mathematical scalars—**integer, real,** and **Boolean.** There are no double-precision types because their use is necessarily machine dependent and one of the goals of Algol was to be a universal, and hence machine-independent, language. Why is double precision machine dependent? Suppose we wanted 10 digits of accuracy in a computation. To decide whether to use single or double precision, we would have to know the word size and floating-point representation used on our computer. If we then transported the program to another computer with a different word size, we would have to look at every declaration to decide whether it should be single or double precision on the new computer. The solution adopted by the Algol designers was to have only one precision in the language; if the *implementation* provided more than one precision, this would be selected by non-Algol constructs (e.g., specially interpreted comments). An alternate approach, which some languages have adopted, is to let the programmer specify the precision desired (e.g., 10 digits), and have the compiler pick the appropriate representation (e.g., single or double). This adds to the complexity of the language since there must be means for the programmer to specify this information. Algol took the simpler approach, a single type **real.**

All these approaches are attempts to follow the Portability Principle:

Portability Principle

Avoid features or facilities that are dependent on a particular computer or a small class of computers.

■ ***Exercise 3-11*:*** Do you agree with the designers of Algol on the issue of precision? Design a mechanism for Algol that allows the programmer to specify the precision of floating-point numbers in a machine-independent way. Discuss the trade-offs involved in each of the two methods.

Algol also did not provide a **complex** data type (recall that FORTRAN provided `COMPLEX` numbers). Although complex numbers are useful in scientific computation, they were omitted from Algol because they are not *primitive*, that is, it is not very difficult or inefficient to define them in terms of the other primitives, namely, **real**s. However, it is not very convenient to have to do so. Consider the following fragment of a FORTRAN program that uses COMPLEX numbers.

```
COMPLEX X, Y, Z
X = (Y + Z)/X
```

To manipulate complex numbers in Algol, it would be necessary to write procedures to perform the complex arithmetic operations (e.g., ComplexAdd, ComplexDivide). Complex numbers themselves would have to be represented either as pairs of real variables or as two-element real arrays. If we chose the latter representation, the FORTRAN fragment shown above would look like this in Algol-60:

```
real array x, y, z, t [1:2];
ComplexAdd (t, y, z);
ComplexDivide (x, t, x);
```

Notice that it is necessary to introduce a new temporary array, t. We can see that the Algol-60 solution is much less convenient and considerably less readable. This is a classic trade-off in programming language design: Are the convenience, readability, and efficiency of an additional data type (**complex**, in this case) worth the complexity of adding it to the language? Notice that there is substantial complexity associated with adding the **complex** type. If we wish to maintain the regularity of the language (thus following the Regularity Principle), we will want complex numbers to be first-class citizens. This means that there must be complex variables, complex arrays, complex parameters, complex operations, and complex relational tests. There must also be a syntax for writing complex numbers and input-output formats for reading and printing them. Finally, it will be necessary to decide whether there are meaningful *coercions* between complex numbers and other values. For instance, we will certainly want to be able to add complex and real numbers. But how about complex numbers and integers? How will we extract the real and imaginary parts of a complex number? We can see that the addition of a data type to a programming language is not a simple decision and should not be made lightly. In this case, the designers of Algol-60 decided to follow the Simplicity Principle, and omit **complex**.

■ ***Exercise 3-12*:*** Do you think complex numbers should have been included in Algol-60? Take a position and defend it.

■ ***Exercise 3-13:*** Program ComplexAdd, etc., in Algol-60 or another language without a built-in **complex** data type.

Sometimes designers do decide to have second-class citizens in a programming language. An example of this is the **string** data type in Algol-60. We mentioned earlier that Algol-60 does not contain any input-output statements such as the FORTRAN READ, WRITE, and FORMAT statements. Rather, it was intended that each Algol implementation would provide a set of library procedures for input-output. The Algol designers saw that this implied a facility for passing *strings* to procedures since otherwise there would be no way of printing headings or other alphanumeric information. Rather than introduce a full-fledged **string** data type with all its associated variable declarations, operators, relations, and so on, the Algol designers decided to allow strings in only a few limited contexts. Specifically, string denotations (i.e., string literals such as `'Carmel'`) are only allowed as actual parameters, and the **string** type is only allowed in the specifications of formal parameters. There are no operators, relations, coercions, or variable declarations for strings. The effect of these restrictions is that the only thing that can be done with strings is to pass them as arguments to procedures. All those procedures can do is pass them as arguments to other procedures, and so forth. What good is this? There's no point in just passing strings around; sooner or later someone must *do* something with them. That is correct, but it cannot be done in Algol. Ultimately, the string must be passed to a procedure coded in some other language, probably assembly language. This would be typical of, for example, input-output procedures.

We have seen that in both FORTRAN and Algol-60 character strings are second-class citizens, although in Algol, at least, this does not cause a loophole in the type system. The exclusion of strings was justified on the basis that these were scientific languages. Commercial programming languages such as COBOL, on the other hand, provided strings as bona-fide data types. We will see, however, the Algol was the last major programming language without strings; almost all newer languages, including PL/I, Algol-68, Pascal, C, and Ada, have some ability to deal with strings in the language.

■ ***Exercise 3-14*:*** Design a **string** data type for Algol-60. Address issues such as the operations, relations, and coercions to be provided.

Algol Follows the Zero-One-Infinity Principle

We have said that *regularity* was a goal of the Algol-60 design. Since a regular design has fewer special cases, it is generally easier to learn, remember, and master. There is a special application of the Regularity Principle called the Zero-One-Infinity Principle.

Zero-One-Infinity Principle

The only reasonable numbers in a programming language design are zero, one, and infinity.

The easiest way to see what this means is to look at a couple of examples. The FORTRAN IV design is filled with numbers other than zero, one, or infinity; for example, identifiers are limited to six characters, there are at most 19 continuation cards, and arrays can have at most three dimensions. Six, 19, three, and so on, are all numbers that a FORTRAN programmer must remember, which will be difficult since the numbers seem completely arbitrary. Consider the FORTRAN limitation of identifiers to six characters. Why six? No one knows, but it probably has something to do with the representation of the symbol table in the original FORTRAN I implementation or the word size on the IBM 704. This number is small enough to be an annoyance to the programmer since a longer identifier will often make the program more readable. The Zero-One-Infinity Principle says that the number of characters allowed in a name should be zero, one, or infinity. Zero, of course, would not make any sense in this case; a one-character limit would make sense, although we would consider it very limiting.

Indeed, many early programming languages (including preliminary FORTRAN and BASIC) allowed only one-character names (as is common in mathematics). Algol and most newer languages have chosen the infinity option; that is, there is no limit on the length of an identifier. In reality there must be *some* limit, such as the memory capacity of the computer, but the limit is so large as to be effectively infinite. In summary, the Zero-One-Infinity Principle states that any limit in a programming language design should be none, one, or (effectively) infinite.

Arrays Are Generalized

Recall that FORTRAN IV arrays are limited to three dimensions (seven in FORTRAN 77) this was a result of efficiency considerations in the first FORTRAN system. This is clearly a violation of the Zero-One-Infinity Principle, so Algol arrays are generalized to allow any number of dimensions. This is not a pointless generalization; arrays of more than three dimensions are not uncommon in scientific computation.

Algol also generalizes FORTRAN arrays by allowing lower bounds to be numbers other than 1. That is, FORTRAN arrays are always addressed as A(1) through A(n), where n is the dimension of the array. In Algol, if the array is declared 'A[m:n]', then the legal array references are A[m] through A[n]. We can see that this array has $n - m + 1$ elements. It is often useful to have lower bounds other than 1. For example, to keep track of number of days according to temperature, from −100 to +200, we declare an array

```
integer array Number of days [-100:200]
```

and then use negative indexes for temperatures below zero. For example, `Number of days [-25]` represents the number of days with a temperature of −25. Of course, the same can be accomplished with a fixed lower bound. `Number of days [temperature]` could be written in FORTRAN as `NUMDAY(TEMPER + 101)`. The latter is less readable and more error-prone since the programmer must always remember to include the *bias* of 101 in every array reference and to correct it if the program changes. Most programming languages designed since Algol-60 allow the programmer to specify both upper and lower bounds.

▩ ***Exercise 3-15:*** Write the addressing equation for one-, two-, and three-dimensional arrays with arbitrary (i.e., user-specified) lower bounds.

▩ ***Exercise 3-16:*** Generalize the above equation for an n-dimensional array with dimensions $[k_1{:}u_1, k_2{:}u_2, \ldots, k_n{:}u_n]$.

Stack Allocation Permits Dynamic Arrays

It is often the case that the size required for an array is not known when the program is written; often it depends on the data with which the program is run. Consider the example in Figure 3.1, the Absolute-Mean program; clearly, the array `Data` should be exactly `N` elements long, where `N` is the number of input values. In the FORTRAN program in Figure 2.3, we dimensioned the array to 900 elements. If there are fewer than 900 input values to be processed, then the rest of the array will be wasted (which means that the program is larger than was necessary). If more than 900 values are supplied, the program will try to oversubscript the array, which either will cause an error or cause the program to fail in some more mysterious way. This is a pervasive problem with languages that have *static* array dimensions; it is always necessary to pick *some* number to use as the dimension. Hence, the programmer must trade off wasting space by making the array large enough to handle almost all applications against not being able to handle some data sets because the array is too small. Static arrays force the programmer to violate the Zero-One-Infinity Principle in the *application* program since in its documentation it will be necessary to say something like, "The number of input values cannot exceed 900."

The Algol committee recognized this problem and devoted much discussion to dynamic arrays between the Algol-58 and Algol-60 reports. We can see from the example in Figure 3.1 that Algol-60 permits expressions to be used in array declarations, `N` in this case. This value `N`, representing the number of input values, is read in within the outer block. The result is that the array `Data` is exactly the right size; there are no unused array elements and there is no limit on the number of input values that can be processed (aside, of course, from the computer's memory capacity). Notice that Algol-60's arrays are dynamic in a limited fashion; the array's dimensions can be recomputed each time the block to which it is local is entered, but once the array is allocated, its bounds remain fixed until its scope is exited. Some other programming languages (including Algol-68) permit arrays that are even more dynamic. In these languages the arrays can grow or shrink at any time; these are sometimes called *flexible* arrays. The Algol-60 design is actually a good trade-off between flexibility and efficiency since, as we will see below, dynamic arrays are very simple to implement on a stack.

In Section 3.3 (pp. 112–114) we saw the way that a stack permits efficient storage allocation in a block-structured language. Every time a block is entered, an activation record for that block, containing all of the block's local storage, is pushed onto the run-time stack. When the block is exited, this activation record is deleted. This makes Algol's dynamic arrays particularly simple to implement; since the array is part of the activation record and its size is known at block-entry time, an appropriate size activation record can be allocated. This activation record is deleted at block-exit time so that on the next entry to that block a completely different size activation record can be created. We can also see some of the difficulty in implementing flexible arrays since, if an array grew, it would be necessary to move everything above it on the stack in order to make room. Algol's dynamic arrays are an excellent example of a programming language design trade-off. Newer languages (e.g., Pascal) have not always provided dynamic arrays; in Chapter 5 we will see some of the problems this has caused.

Algol Has Strong Typing

Algol-60 is an example of a programming language with *strong typing*, which means that the type abstractions of the language are enforced. A strong type system prevents the programmer from performing meaningless operations on data. A less formal way of saying this is that there are no typing "loopholes"; for instance, a programmer can't do an integer addition on floating-point numbers, or do a floating-point multiply on Boolean values.[3] In Chapter 2 we saw that FORTRAN is not a strongly typed language. In particular, by using `COM`–

[3] Algol does have a loophole in its typing caused by inadequate specification of procedural parameters. Since Pascal has the same problem, we defer its discussion to Section 5.5

MON and EQUIVALENCE it is possible to set up situations where two or more variables, with different types, are *aliases* for the same location. This is a security problem when done accidently and a maintenance problem when done intentionally. In Algol there is no way to trick the system into believing and acting as though an integer were a Boolean or anything similar.

We should carefully distinguish between illegitimate type violations of this sort and perfectly legitimate *conversions* and *coercions* between types. For instance, Algol, like most programming languages, coerces integers to reals and provides a conversion operator for converting reals to integers. These are machine-independent operations. The results of a type system violation, however, depend on the particular data representations used on a particular implementation; we say they are *implementation dependent.* Obviously, implementation dependencies violate the Portability Principle.

If you have done any system programming using a high-level language, you are probably saying right now, "That is all very well for application programmers, but systems programmers often *have* to violate the type system." That is correct. For example, if we are programming a memory management system, it is necessary to treat memory cells as raw storage without regard for the type of the values stored in them. As another example, the input-output conversion routines will probably have to be able to manipulate the characteristic and mantissa of floating-point numbers as integers. However, we must recognize that a programming language's type system is a *safety feature*, and as such, it must be circumvented with extreme care. You are probably aware of the interlock on most electrical equipment that disconnects the line cord when the back is removed; this is to prevent dangerous electrical shock. Since electrical technicians must be able to operate the equipment with its back off in order to repair it, they use "cheater cords" to supply power and "cheat" the interlock system. This requires extra precautions that are part of the training of an electrical technician. The same applies to the type system; it is true that it must be "cheated" in some situations, but this should be done only by "qualified service personnel" who know the proper precautions. One of the precautions is that it be done only when *really* necessary; another is that the violation be clearly documented. Since we all consider ourselves qualified, and we all think that our needs are *really* necessary, considerable self-discipline is required. In fact, most legitimate type system violations can be replaced by special conversion functions that provide access to the representation. Most of the violations that people think are necessary really are not; there are usually better, safer ways to accomplish the task. Most programming languages that are intended for systems programming do provide some "loophole" through the type system, but the conscientious programmer is advised to avoid it if at all possible, since it will likely lead to unreliable, nonportable, unmaintainable programs. Intentional cheating of this safety system will not be discussed further in this book.

▩ ***Exercise 3-17*:*** Do you agree with the above analysis of violations of the type system? List some of the situations in which a violation of the type system is justified. Describe safe programming language mechanisms that will handle these problems. How much additional complexity does this add to the programming language? (Keep in mind that these facilities will be rarely used.)

▩ ***Exercise 3-18*:*** Most FORTRAN systems do not check that the types of actual and formal parameters agree. For example, an integer can be passed to a subprogram that is expecting 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.

Procedures Are Recursive

To illustrate the idea of a recursive procedure, we use a very simple example. In fact, this particular function could be as easily defined without recursion. The *factorial* $n!$ of an integer n is defined to be the product of the integers from 1 to n. That is,

$$n! = n(n-1)(n-2)\ldots(2)(1)$$

This is not an entirely precise definition since it is not clear about the meaning of $n!$ for $n < 3$. Rather, we can define $0!$ to be 1 and $n!$ to be $n \times (n-1)!$ for $n > 0$. This is a preferable definition since it clearly specifies the meaning of $n!$ for all $n \geq 0$. This can be summarized in a *recursive* definition such as

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

This is a recursive definition because the thing being defined is defined in terms of itself (recur = to happen again). Of course, this would not be a sensible thing to do if there were not some *base* for the recursion, as is provided by the $n = 0$ case in the definition of factorial. Recursive definitions are very common in computer science and mathematics so it is important to get used to them.

Algol permits procedures to be defined recursively, as we can see by looking at an Algol factorial procedure:

```
integer procedure fac(n);
  value n; integer n;
  fac := if n = 0 then 1 else n × fac(n - 1);
```

You may be surprised to see an **if** used on the right of an assignment statement; this is called a *conditional expression* (as opposed to the usual *conditional statement*). The condition is used to determine whether the consequent (**then**-part) or alternate (**else**-part) is to be evaluated. This is a common pattern for many recursive definitions: A *stopping condition* is used to break the evaluation into two cases, one that involves a recursive call of the procedure and one that does not. We will see many examples of recursive definitions in later chapters.

■ ***Exercise 3-22:*** The number of combinations of m things taken k at a time, $C(m,k)$, is defined by

$$C(m,k) = \frac{m!}{k!(m-k)!}$$

Devise a recursive definition of $C(m,k)$ in terms of $C(m-1,k)$ and write an Algol procedure to implement it.

Locals May Have Several Instantiations

We saw in Chapter 2 that the values of actual parameters are stored in memory locations in the activation record for the procedure. There is one of these locations for each formal parameter, and whenever a formal parameter is referenced, the corresponding location is accessed. Now consider the factorial procedure described above; there must be a memory *lo*cation corresponding to `n` to hold the value of that parameter. Therefore, when `fac(3)` is invoked, this location will contain the number 3. Before this invocation of `fac` is completed, the expression `fac(2)` must be evaluated. We can see that if there were only one location for `n`, the value 2 would overwrite the value 3 before the evaluation of `fac(3)` had been completed. The same would happen again as `fac(1)` and `fac(0)` were invoked. This is clearly wrong; it is necessary to set aside separate memory locations corresponding to `n` for each invocation of `fac`. Another way to say this is that each invocation of `fac` must have its own *instance* of the formal parameter `n`, which will contain the actual parameter to that invocation. We will see in Chapter 6 that this is accomplished by creating a new activation record for `fac` each time it is called. In general, each invocation (or *activation*) of a procedure results in the *instantiation* (i.e., creation) of a new activation record for that procedure. These activation records contain space for the parameters and local variables of the procedure. We have already seen (Section 3.4) how the creation of activation records at block-entry time permits efficient dynamic storage management. We will see in Chapter 6 how the instantiation of activation records is accomplished.

Parameters Can Be Passed by Value

Our definition of the `fac` procedure contained the specification **value** `n`. This specifies that the parameter `n` is to be *passed by value*, one of the two ways of passing parameters in Algol-60. As its name suggests, pass by value is the first half of pass by value-result; that is, the value of the actual parameter is copied into a variable corresponding to the formal parameter. Pass by value is quite secure. Since a local copy is made of the actual, there is no possibility of an assignment to the formal overwriting the actual; such an assignment will just modify the local copy. For example, a procedure that is similar to the one that allowed us to change the value of constants in FORTRAN:

```
procedure switch (n);
  value n; integer n;
  n := 3;
```

is perfectly harmless in Algol. An invocation such as `switch (2)` will cause the value 2 to be copied into the local storage for the formal `n`. The assignment `n := 3` merely alters the value of this location; it has no effect on the actual, 2, and does not endanger the literal table. You probably have noticed that **value**-parameters are useful only for input parameters; they do not allow a value to be output from a procedure. For example, the call `switch (k)` can have no effect on the variable `k`. Later in this section, we investigate Algol's other parameter passing mode, which does allow output parameters.

Pass by Value Is Very Inefficient for Arrays

Consider the following skeleton for a procedure to compute the average of an array:

```
real procedure avg (A, n);
  value A, n;
  real array A; integer n;
  begin
    .
    .
    .
  end;
```

Since both the array to be averaged (A) and the number of elements in the array (n) are input parameters, we have passed them both by value. Let's consider the consequences of this. When the procedure avg is called, an activation record will be created containing space for the values of the actual parameters. This will include space for the entire array A! Not only is this wasteful of storage, but it also wastes time since the entire actual parameter corresponding to A must be copied into the activation record! For a large array, it could take almost as long to pass the array parameter as to compute its average. The conclusion we can draw is that pass by value is not a satisfactory way of passing arrays.

■ ***Exercise 3.23*:*** Suggest an alternative to Algol-60's pass by value that has the security advantages of pass by value (i.e., the value of input parameters cannot be changed) but is more efficient for arrays.

Pass by Name Is Based on Substitution

We have seen in Chapter 2 that the problem with FORTRAN's parameters results from the failure to distinguish parameters intended for input from those intended for output (or both input and output). Algol-60 attempted to solve this problem by providing two parameter passing modes: pass by value for input parameters and *pass by name* for all other kinds of parameters. Suppose we wanted to write a procedure to increment a variable; for example, Inc (i) adds 1 to i. We can pattern it after a FORTRAN procedure:

```
procedure Inc (n);
  integer n;
  n := n + 1;
```

If the parameter n were passed by value, this would not work since the assignment n := n + 1 would affect the local copy of the actual but not the actual 'i' itself. What we want is for the *name* 'i' to be substituted into the procedure rather than the *value* of 'i'. This is accomplished by pass by name, which is the mode we get if we do not specify pass by value. Thus, in the above example 'n' is passed by name.

In Chapter 2 (Section 2.3) we discussed the *Copy Rule* for procedure invocation. The rules states that a procedure can be replaced by its body with the actuals substituted for the formals. This is the effect of pass by name in Algol-60, so the call Inc (i) has the same effect as if it were replaced by

```
i := i + 1;
```

that is, the body of Inc with 'n' replaced by 'i'. Similarly, the invocation Inc (A[k]) acts as though it were replaced by

```
A[k] := A[k] + 1;
```

We can see that pass by name satisfies Algol's need for output parameters. Although name parameters *act* as though they were substituted for the corresponding formals, this is not the way they are actually implemented; it would be too inefficient. The implementation of name parameters is discussed later in this section.

Pass by Name Is Powerful

Consider the following FORTRAN subroutine:

```
SUBROUTINE S (EL, K)
K = 2
EL = 0
RETURN
END
```

and the program segment:

```
A(1) = A(2) = 1
I = 1
CALL S (A(I),I)
```

Further, suppose that parameters are passed by reference. When `S` is called, a reference must be passed for each of the actual parameters `A(I)` and `I`. Since `I` has the value 1 at the time of call, the reference passed for `A(I)` will be the address of `A(1)`. The subroutine `S` has two effects. First, it assigns `K = 2`, which assigns 2 to `I` since the formal `K` is bound to the address of `I`. Second, it assigns `EL = 0`, which assigns 0 to `A(1)` since the formal `EL` is bound to the address of `A(1)`. The fact that in the meantime `I` has been changed to 2 has no effect on `EL` since the reference of `A(I)` was computed at call time. When `S` exits, `I` will have the value 2 and `A` will have the values (0,1).

Next, consider an analogous Algol program:

```
procedure S (el, k);
  integer el, k;
  begin
    k := 2;
    el := 0
  end;
A[1] := A[2] := 1;
i := 1;
S (A[i], i);
```

Since the parameters are passed by name, the effect of `S` will be as though the actuals were substituted for the formals. That is, the effect of `S (A[i], i)` will be

```
i := 2;
A[i] := 0;
```

Thus, `S` will assign 2 to `i` and 0 to `A[2]`, so when `S` exits `i` will have the value 2 and `A` will have the value (1,0). Notice that the Algol result is different from the FORTRAN result. It is as though the formal parameters `el` were bound to the string `A[i]` rather than an address. Therefore, every time `el` is referenced, the actual parameter is reevaluated. Hence, if `i` has changed in the meantime, a different element of `A` will be referenced.

Jensen's device makes explicit use of this property of name parameters. Suppose we wish to write a procedure to implement the mathematical sigma notation. That is, we want a procedure `Sum` such that

$$x = \sum_{i=1}^{n} V_i$$

can be written

```
x := Sum (i, 1, n, V[i])
```

This is easy to do with name parameters; by altering the index variable `i`, the actual `V[i]` can be made to refer to each element of the array. This is Jensen's device. For example,

```
real procedure Sum (k, l, u, ak);
  value l, u;
  integer k, l, u; real ak;
  begin real S; S := 0;
    for k := l step 1 until u do
      S := S + ak;
    Sum := S;
  end;
```

To determine the effect of

```
x := Sum (i, 1, n, V[i])
```

simply perform the substitutions into the body of `Sum`. The result is

```
begin real S; S := 0;
  for i := 1 step 1 until n do
    S := S + V[i];
  x := S
end
```

We can see that it has the desired effect since the **for**-loop alters the values of `i` from 1 to `n` and this causes `V[i]` to refer to `V[1]` through `V[n]`. This `Sum` procedure is very general; for instance,

$$x = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}$$

can be computed by

```
x := Sum (i, 1, m, Sum (j, 1, n, A[i,j]))
```

There are few programming languages that provide this flexibility.

How is this powerful feature implemented? It would be much too inefficient to follow the Copy Rule literally: This would involve passing the text of the actual parameter to the procedure. It would then be necessary to compile and execute this text every time the parameter was referenced. A better solution would be to compile the actual parameter into machine code and then copy this code into the callee everywhere the parameter is referenced. This would also be inefficient since the code for the actual parameter would be copied many times. It would be difficult to implement since different actual parameters would have different-length code sequences. These problems can be avoided by passing the address of the code sequence compiled for the actual parameter. Then, every time the parameter is referenced, the callee can execute the code for the parameter by jumping to this address. The result of executing the code, an address of a variable or array element, is then returned to the callee. This implementation technique has the proper effect of reevaluating the actual parameter every time it is referenced. It is also reasonably efficient.

One of the first Algol implements, P. Z. Ingerman, used the name *thunk* to refer to these pieces of code that provide an address. The name has stuck.

Thunks are very similar to procedures. Like procedures, they are called from several different locations and must return a result to their callers. In fact, a simple way of implementing name parameters is to convert the actual parameters into procedures and to convert references to the formal parameters into indirect procedure calls. For example, in the call

```
x := Sum (i, 1, m, Sum (j, 1, n, A[i,j]))
```

the actual parameter `Sum (j, 1, n, A[i,j])` would be converted into a parameterless function; the address of this function would then be passed to the outer activation of `Sum`. Within `Sum`, references to the formal parameter `ak` would be interpreted as indirect invocations of this procedure. Thus, we formally define a *thunk* to be an invisible, parameterless, address-returning function used by a compiler to implement name parameters and similar constructs.

■ ***Exercise 3-24:*** Write an Algol statement to compute

$$x = \sum_{i=1}^{n} A_{ij}$$

■ ***Exercise 3-25:*** Write an Algol statement to compute

$$x = \sum_{i=1}^{m} \sum_{j=1}^{i} (j!\, A_{ji})$$

Pass by Name Is Dangerous and Expensive

You may think that pass by name is the ideal solution to the output parameter problem. The idea of substitution, on which it is based, is very simple and we get flexibility like the `Sum` procedure for free in the bargain. Unfortunately, pass by name has some dark corners and hidden inefficiencies. To show this, we will consider what should be a simple use of name parameters—an exchange procedure. The idea is to write a procedure `Swap (x,y)` that exchanges the values of the variables `x` and `y`. This definition would seem to do it:

```
procedure Swap (x, y);
  integer x, y;
  begin integer t;
    t := x;
    x := y;
    y := t
  end
```

To see if this works, we can try `Swap (i,j)` and use the Copy Rule:

```
begin integer t;
  t := i;
  i := j;
  j := t
end;
```

As we would expect from a swap procedure, `Swap (j,i)` produces exactly the same effect as `Swap (i,j)`. Now let's consider the effect of `Swap (A[i],i)`:

```
begin integer t;
  t := A[i];
  A[i] := i;
  i := t
end;
```

This is correct. Now consider `Swap (i,A[i])`, which should have the same effect:

```
begin integer t;
  t := i;
  i := A[i];
  A[i] := t
end;
```

This does something completely different! If you do not see this, then try executing it assuming that `i` contains 1 and `A[1]` contains 27. The effect will be to assign the value of `A[1]` to `i` and the value of `i` to `A[27]`; it does not do an exchange at all! It is a sign of a design mistake when a simple procedure, such as `Swap`, has such surprising properties. We have programmed this procedure in the obvious way and found that it does not work. What is even worse is that computer scientists have shown that there is *no way* to define a `Swap` procedure in Algol-60 that works for all actual parameters! Lest you think that this reveals some hidden subtlety in the idea of a swap, we must hasten to say that it is trivial to write a correct `Swap` procedure in FORTRAN:

```
SUBROUTINE SWAP(X,Y)
INTEGER X, Y, T
T = X
X = Y
Y = T
```

```
RETURN
END
```

We can see why this is so: In the FORTRAN procedure X and Y are bound to fixed locations at call time; in the Algol procedure the parameters x and y are reevaluated every time they are used and, hence, may refer to different locations on each use. It is traps such as these that have led language designers to avoid pass by name in almost all languages designed after Algol-60. We will see in Chapter 6 that pass by name is also quite expensive to implement compared to pass by reference.

The three parameter passing modes that we have discussed can be distinguished by the times at which they inspect the value of the actual.

1. *Pass by value.* At the time of call, the formal is bound to the *value* of the actual. Since the value parameter takes a snapshot of the actual parameter, later changes in the actual's value will not be seen by the callee.
2. *Pass by reference.* At the time of call, the formal is bound to a reference to the actual. The reference cannot vary thereafter, although the value stored at that reference can.
3. *Pass by name.* At the time of call, the formal is bound to a thunk for the actual. Although this cannot vary, each time it is evaluated it may return a different reference and consequently a different value.

We can see that pass by value represents an early inspection time, while pass by name represents a very late inspection time (the latest found in most languages). A consequence of this is that there is less variability and flexibility in pass by value than in pass by name. One way to think of this is that a late inspection time, such as pass by name, puts off until later a commitment to the meaning of the formal parameter.

▩ ***Exercise 3-26**:*** Propose an alternate to Algol's pass by name. It should allow values to be output through the parameters of a procedure. Try to make it powerful enough to define a Sum procedure without it also having the problems of pass by name.

▩ ***Exercise 3-27:*** Algol makes pass by name the *default* mode for passing parameters; that is, it is what you get if you do not include a **value** specification. Given that both value and name parameters have their problems, discuss which should be the default parameter passing mode.

Good Conceptual Models Help Users

Before leaving the topic of name parameters, we consider the Copy Rule as an example of a *conceptual model.* A conceptual model is a mental representation of the way something works; it helps us to predict the effects of our actions when dealing with that thing. We all have conceptual models, of simple objects such as doors and bicycles, and of complex objects such as VCRs and computers. Some of these models are acquired implicitly through experience; others are learned explicitly from user's manuals, classes, or other explanations. According to psychologist Donald Norman, "A good conceptual model allows us to predict the effects of our actions," but the Copy Rule shows that a model may be good without being factually correct. As we have seen, the Copy Rule allows us to predict the effects of passing parameters by name, although it is not an accurate description of how name parameters are implemented. Indeed, an accurate description of the implementation of name parameters (involving creation of thunks and passing procedural parameters) would not be a very good conceptual model, because it is difficult to understand, which makes it less useful for predicting the results of actions.

Norman distinguishes three conceptual models (there may be more): the *designer's model*, the *system image*, and the *user's model.* The designer's model usually reflects the system's construction, which for a programming language feature includes its implementation, formal syntax, etc. The system image is created by the designer and is the basis on which the user's image is formed. In addition to the system itself (which may present various displays and controls, or even expose its mechanism to the user's view), the system image includes manuals, diagrams, explanations, instruction, etc. provided with the system. Users form their own models on the basis of the system image, and their competence and comfort in using the system will depend on the quality of those models. Thus the system image is the primary medium for communicating models from the designer to the users.

It is the designers' responsibility to invent good conceptual models for their systems, that is, models that are understandable yet accurate enough to facilitate correct prediction, and to embody them in the system image. If a good conceptual model cannot be invented, it is a bad design, for it is a system that will be difficult and unreliable to use. Programming language designers (indeed, designers of all computer systems) would do well to read Donald A. Norman's *Psychology of Everyday Things* (Basic Books, 1988) to understand the critical role of conceptual models in the usability of artifacts.

Out-of-Block gotos Can Be Expensive

We saw that FORTRAN has different scope rules for different kinds of names. For example, variables and statement labels are subprogram local while subprogram names are global. Algol has one scope rule for all identifiers: An identifier declaration is visible if we are nested within the block in which it occurs (and the same identifier hasn't been redeclared in an inner, surrounding block). This is reflected in the contour diagrams. A consequence of this is that we can jump to statement labels from any block nested in the block in which they are declared. Consider this program skeleton:

```
a: begin array X[1:100];
          .
          .
          .
          b: begin array Y[1:100];
                 .
                 .
                 .
                 goto exit;
                 .
                 .
                 .
             end;
exit: .
      .
      .
   end
```

Since the label `exit` is declared in the block (a), which encloses the block (b), this label is visible to the **goto**-statement in the block (b). Consider the effect of executing the **goto**-statement. When this occurs there will be activation records for both blocks (a) and (b) on the stack. If we exited block (b) normally (i.e., through the **end**), the activation record for (b) would be deleted, leaving that for (a) on the top of the stack. The same must be done if block (b) is exited by a jump: The activation record for (b) must be deleted. Therefore, the process of executing an Algol **goto**-statement may be much more complicated than a simple machine jump since it may involve exiting through several levels of block structure.

The problem is more complicated than this since the number of levels to be exited (and hence the number of activation records to be popped) may not even be a constant. Consider this Algol program:

```
begin
  procedure P(n);
    value n; integer n;
    if n = 0 then goto out
    else P(n-1);

  P(25);
out:
end
```

We can see that `P(25)` will cause `P` to be called recursively 25 times, at which point it will jump to the label `out`. Each recursive call will require an activation record for `P` to be pushed onto the stack, all of which must be deleted when the procedure transfers to `out`. Since the number of recursions, and hence the number of activation records to be discarded, are data dependent, the compiler cannot know beforehand how many activation records to delete. Instead, this **goto**-statement might be implemented as a call of a run-time routine that finds the appropriate activation record and deletes all the ones above it. Needless to say, this can be an expensive process.

Feature Interaction Is a Difficult Design Problem

Feature interaction is one of the most difficult problems in language design. In our example, two apparently simple features—the Algol visibility rules and the **goto**-statement—interact in a way that is complex and can lead to inefficient execution. This is a very common problem in language design since the number of possible interactions is so large. If there are 100 features in a language (a small number), then the designer must investigate $100^2 = 10{,}000$ interactions between pairs of features, $100^3 = 1{,}000{,}000$ interactions between triples of features, and so on. Since it is impossible to investigate *all* of these possible interactions, successful language designers must (through experience) develop a "nose" for spotting possible problem areas. This process is greatly aided if the language has a regular, orthogonal structure. Since the number of interactions between even pairs of features goes up with the square of the number of features in the language, we can see that a *simple* language is much 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.

▩ ***Exercise 3-30*:*** Define a simplified looping construct (or constructs) for Algol. How will you respond to those users that say they need some of the baroque features you have eliminated?

The switch Is for Handling Cases

In Chapter 2 we discussed FORTRAN's computed GOTO-statement, which was a mechanism for handling several different cases of a problem. Just as it did with FORTRAN's DO-loop, Algol has generalized FORTRAN's computed GOTO. The construct in question is called a **switch**-*declaration* and amounts to an array of statement labels. Let's consider an example. Suppose we are processing personnel records and must handle separately the cases of single, married, divorced, and widowed employees. Further, suppose that these cases are represented by the numbers 1, 2, 3, 4. The four cases can be handled by a **switch**-declaration such as

```
begin
  switch marital status = single, married, divorced, widowed;
    .
    .
    .
  goto marital status[i];
single:     ... handle single case ...
            goto done;
married:    ... handle married case ...
            goto done;
divorced:   ... handle divorced case ...
            goto done;
widowed:    ... handle widowed case ...
done:       ...
end
```

It is as though marital status were an array initialized to the statement labels single, married, divorced, and widowed. The **goto**-statement then transfers to the label given by marital status[i].

In the above example, the flow of control can go in one of four directions, depending on the value of i. These four control paths rejoin at the label done, which is a good pattern for using a **switch** since it breaks the processing down into four distinct cases. This pattern is not required by the **switch**, however, and it is possible to use it to write programs that are very hard to understand. Earlier in this chapter we discussed some of the problems with the **goto**-statement, in particular, that it complicated correlating the dynamic and static structure of the program. We can see that the **switch** is even worse since it is not even obvious where it goes! There are several possible destinations; they may be almost anywhere in the program, and the list of them is in the **switch**-declaration, which may be very far away from the **goto**. These are some of the reasons the **switch** has been replaced in newer languages, such as Pascal and Ada, by the much more structured **case**-statement.

The switch Is Baroque

The **switch** is another example of a baroque feature, a fascination-driven extrapolation of the computed GOTO. For instance, Algol permits the **switch** elements to be conditional expressions that are evaluated at the time of the **goto**. Consider the following example:

```
begin
  switch S = L, if i > 0 then M else N, Q;
  goto S[j];
end
```

If j = 1, then the **goto** transfers to label L; if j = 3, then the **goto** transfers to label Q; and if j = 2, the **goto** transfers to label M or N, depending on the value of i. We can see this by the Copy Rule, that is, by substituting the value of S[2] into the **goto**-statement:

```
goto if i > 0 then M else N;
```

This will transfer to M if i > 0, and N otherwise. This textual substitution process may remind you of name parameters; in fact, they are essentially the same and they must be implemented in the same way, by thunks. They also have many of the same problems as name parameters, which does not help an already difficult-to-understand feature.

■ ***Exercise 3-31*:*** The effect of an Algol statement such as **goto** S[k], where k is out of range (i.e., either less than 1 or greater than the number of **switch** elements in S), is to go on to the following statement. In other words, an out-of-range **switch** is interpreted as a *fall-through.* Evaluate this behavior with respect to the Security Principle. Mention any situations in which this behavior would be desirable.

■ ***Exercise 3-32*:*** What restrictions would you place on the **switch**-declaration to make it easier to understand and implement, without seriously diminishing its usefulness?

EXERCISES*

1.** Study the Algol-68 language and evaluate its control structures with respect to regularity, orthogonality, and structure.

2. Read about **own** variables in the Algol-60 Report (or an Algol text). Investigate whether they can be used to solve the information-hiding problems we encountered in the symbol table example in Section 3.3.

3.** In Algol-60 arrays as well as simple variables can be **own**. Arrays can also be dynamic. Discuss feature interaction in dynamic **own** arrays. What does it mean when an **own** array changes size? How might dynamic **own** arrays be implemented?

4. Design a **complex** data type for Algol. Discuss operators, relations, coercions, declarations, etc.

5. Investigate the use of Algol-60 for systems programming. What problems would be encountered? Can they be avoided or should Algol be extended to solve them?

6. Describe how strings could be made first-class citizens in Algol-60. Describe in detail data types, operators, relations, coercions, etc.

7. Read, summarize, and critique Donald Knuth's article, "Structured Programming with GOTO Statements" (*ACM Comp. Surveys 6*, 4, 1974).

8. Write a recursive procedure to compute the Fibonacci numbers. Recall that $F(0) = F(1) = 1$ and $F(n) = F(n - 1) + F(n - 2)$ for $n > 1$. Compute the asymptotical time complexity of this procedure and compare with the asymptotical complexity of the corresponding iterative procedure. If the recursive solution is much worse than the iterative one, then define a more efficient recursive procedure.

9. Prove that it is impossible to write in Algol-60 a Swap procedure that will work for all **integer** operands.

10. What values are printed by the following Algol program if
a. x is passed by value and y is passed by value;
b. x is passed by value and y is passed by name;
c. x is passed by name and y is passed by value; and
d. x is passed by name and y is passed by name?

```
begin integer i, j; integer array A[1:3];
   procedure P(x,y); integer x,y;
      begin
         y := 2;
         Print(x);
         i := 3;
         Print(x);
         i := 3;
         Print(x)
         Print(y)
      end;
   A[1] := 7;   A[2] := 11;   A[3] := 13;
   i := 1;
   P (A[i], i);
   P (i, A[i])
end
```