5 RETURN TO SIMPLICITY: PASCAL

5.1 HISTORY AND MOTIVATION

There Were Many Attempts to Extend Algol's Applicability

Algol was so successful that there were immediate attempts to use its ideas in other application areas. These included Algol-like languages for list processing, string manipulation, systems programming, and artificial intelligence. There were several proposals to include the input-output facilities that Algol lacked and that might have made it feasible to use Algol for commercial programming.

PL/I Attempted To Be One Language for All Applications

PL/I, as we saw in Chapter 2, was IBM's 1964 effort to design FORTRAN VI. The resulting language, which incorporated ideas from FORTRAN, Algol, and COBOL, was so different from FORTRAN that it was renamed PL/I (programming language one). PL/I combined ideas from many sources, including the block structure of Algol, the record and file handling capabilities of COBOL, and the syntactic style of FORTRAN. The result was a very large language—perhaps the largest ever to achieve widespread use. It is more like the union of FORTRAN, COBOL, and Algol than their intersection. The promoters of PL/I argued that users could learn just the subset of the features that were relevant to their applications. This turned out not to be possible since the subtle and unpredictable interactions among the various features of PL/I meant that, practically, programmers had to be aware of the entire language. As a result, PL/I has become the classic example of a "Swiss army knife" approach to language design; that is, the attempt to provide in one language all the gadgets or features that anyone might ever want. The result is inevitably a language so large as to be unmanageable and too complicated to be mastered by most programmers. Dijkstra, Hoare, and several other computer scientists have severely criticized PL/I; in Chapter 2 we cited Dijkstra's characterization of PL/I as a fatal disease. In his Turing Award paper, he called PL/I "a programming language for which the defining documentation is of a frightening size and complexity. Using PL/I must be like flying a plane with 7000 buttons, switches, and handles to manipulate in the cockpit." This is an example of fear resulting from a loss of control and from the excessive power of an overelaborated tool (Section 1.4).

Extensible Languages Were Another Approach

Many programming language designers believed that it was futile to design a language that was all things to all programmers. After all, we do not try and make do with one camera that is suitable for everyone and everything—for professionals, amateurs, beginners; for science, art, astronomy, advertising, and so forth. Rather, these designers believed that a more effective approach was to have a closely knit family of application-oriented languages. Of course, the cost of implementing all of these languages would be prohibitive, so it was necessary to develop a new approach to language implementation. This was the *extensible language* approach; the idea that one is given a simple, application-independent *kernel*, or *base*, language and an *extension mechanism* that allows the kernel language to be extended into various application areas. Early examples of extensible languages are the MAD language and McIlroy's "syntax macros."

A Simple Kernel Language Was Necessary

The kernel language had to be simple and efficient since it would form the basis for all the application-oriented languages. Further, it had to be application independent so that it would be useful for implementing extended languages in a wide variety of application areas. The most common choices for kernel languages were simple subsets of Algol-60 (with somewhat more general data structuring capabilities). Since, as opposed to PL/I, the goal was to have as small a kernel language as possible, the extensible language effort produced useful experience about the minimal facilities required in a usable language.

There Were Many Kinds of Extension

Extensible languages varied greatly in the forms of extension they provided. A form common to almost all these languages was *operator extension*, the ability to define new, application-oriented operators. For example, if a programmer wanted to use `x # y` for the symmetric difference between two real numbers, a typical operator definition would be

```
operator 2 x # y; value x, y; real x, y;
  begin
    return abs(x-y)
  end;
```

We can see that this is a lot like an Algol procedure declaration, except that the template `x # y` is used in place of a procedure heading like `dif (x, y)`. The `2` following **operator** indicates the precedence of the new operator; for instance, `1` might be the precedence of the relational operators (=, <, >, etc.), and `3` might be the precedence of the additive operators (+, −).

Other languages went much further by providing *syntax macros* that allowed the programmer to introduce new syntax into the programming language. For example, a summation notation might be defined.

```
real syntax sum from i = lb to ub of elem;
  value lb, ub;
  integer i, lb, ub; real elem;
  begin real s; s := 0;
    for i := lb step 1 until ub do
      s := s + elem;
    return s
  end;
```

In this case, the syntactic template replaces the usual heading, such as `sum (i, lb, ub, elem)`. This definition allows the programmer to write statements such as

```
Total := sum from k = 1 to N of Wages[k];
```

instead of the usual

```
Total := sum (k, 1, N, Wages[k]);
```

The intention was that programmers could use the notations familiar in their application areas.

Extensible Languages Are Usually Inefficient

Unfortunately, extensibility usually resulted in comparatively inefficient languages. First, the necessity of handling a variable syntax made extensible compilers large and unreliable. Second, the fact that all source constructs were ultimately reduced to kernel language constructs meant that minor inefficiencies in the kernel implementation became magnified at the application language level. Part of this inefficiency is unavoidable since there is always a small overhead associated with the concatenation of kernel language constructs that might have been avoided by implementing the extended facilities directly.

Extensible Languages Have Poor Diagnostics

Another problem with extensible languages was their poor diagnostics. Since most of the error checking (for instance, checking for type compatibility) was done by the kernel language compiler, most diagnostics were issued in terms of kernel language constructs. This loss of transparency was, of course, confusing for users working at the application language level.

These problems, and others, ultimately defeated extensible languages. Although they were once the most active area of programming language research, they are now rarely discussed. This does not imply that all of their ideas have been abandoned; we will see that the idea of extensible data types has been incorporated into Pascal and its successors and that a limited form of operator extension is provided by Ada (Chapters 7 and 8).

Wirth Designed a Successor to Algol-60

After the release of the Revised Report on Algol-60, the Algol committee continued to meet in order to develop a successor to Algol-60. Niklaus Wirth and C. A. R. Hoare, in their article "A Contribution to the Development of Algol" (in the *Communications of the ACM*), had already suggested several modest but important improvements to Algol-60. In 1965 these ideas were presented to the Algol committee, which rejected them in favor of the larger, more subtle, excessively complex language now known as Algol-68.

The language that Wirth presented was implemented at Stanford University and became known as Algol-W. It was used as an instructional language at Stanford and several other universities for many years. In the meantime, Wirth had designed and implemented two other programming languages—Euler and PL360—which, like Algol-W, were characterized by their extreme simplicity.

A Competitor to FORTRAN Must Have Clear Advantages

The experience with first- and second-generation languages (such as FORTRAN and Algol-60) had led to the belief that useful languages with powerful facilities were inefficient at both compile-time and execution time. Further, there was the long-standing belief that different languages were needed for commercial and scientific programming, a belief that was reinforced by the failure of PL/I. Wirth knew that if a new language were to be a significant competitor to FORTRAN, it would have to have clear advantages, such as the ability to handle nonnumeric data, and at the same time maintain the compile-time and run-time efficiency of FORTRAN. Wirth saw that it was Algol's data types that limited its applications to scientific applications, so his approach was to start with Algol-60, eliminate the ill-conceived or expensive features, and expand the data structuring capabilities while maintaining its efficiency. These considerations were the motivation for the Pascal language.

Pascal Combines Simplicity and Generality

The Pascal design had explicitly stated goals:

1. The language should be suitable for teaching programming in a systematic way.
2. The implementation of the language should be reliable and efficient, at compile-time and run-time, on available computers.

The development of Pascal began in 1968 and resulted in a compiler written entirely in Pascal in 1970. The Pascal Report's brevity, 29 pages, compares favorably with the Algol-60 Report's 16 pages. The language was slightly revised in 1972 and became an international standard in 1982.[1] The spread of Pascal was aided by the development in 1973 of a portable Pascal system comprising a compiler (written in Pascal and generating P-code, a

[1] See the International Organization for Standards' *Specification for Computer Programming Language Pascal*, ISO 7185-1982, 1983.

pseudo-code designed for Pascal) and a P-code interpreter written in Pascal. In particular, the availability of this system encouraged the use of Pascal on microcomputers in the late 1970s and early 1980s. Pascal has become very popular as a language for teaching programming and is widely used on microcomputers. Therefore we will take it as an example of a third-generation programming language.

5.2 DESIGN: STRUCTURAL ORGANIZATION

Pascal's Syntax Is Algol-Like

Figure 5.1 displays a small Pascal program to compute the mean of the absolute value of an array. Notice that the general style of Pascal is very similar to Algol; it is an *Algol-like* language. In this chapter we will show all of the reserved symbols of Pascal in boldface. This will simplify distinguishing built-in and user-defined constructs. Unlike Algol's keywords, Pascal's reserved words are not typed differently from identifiers.

```
program AbsMean (input, output);
  const Max = 900;
  type index = 1 .. Max;
  var
    N: 0 .. Max;
    Data: array [index] of real;
    sum, avg, val: real;
    i: index;
begin
  sum := 0;
  readln(N);

  for i := 1 to N do
    begin
      readln (val);
      if val < 0 then Data[i] := -val
      else Data[i] := val
    end;

  for i := 1 to N do
    sum := sum + Data[i];

  avg := sum/N;
  writeln (avg)
end.
```

Figure 5.1 A Pascal Program

There Are New Name, Data, and Control Structures

Pascal includes important additions to Algol's name, data, and control-structuring mechanisms. We can see examples of variable declarations in Figure 5.1; they are introduced by the word **var** and have the syntax

⟨names⟩ : ⟨type⟩;

Procedure and function declarations are quite similar to Algol's except that the **begin** comes after the local declarations rather than before them:

```
procedure ⟨name⟩ (⟨formals⟩);
  ⟨declarations⟩
begin
  ⟨statements⟩
end;
```

In addition to variable and procedure declarations, Pascal has constant and type bindings. We can see this in Figure 5.1; `Max` is declared to be a constant equal to 900, and `index` is declared to be the type of integers in the range 1 to `MAX`. This new data type is then used in the declarations of the array `Data` and the variable `i`. Type declarations are introduced by the word **type** and have the syntax

```
⟨name⟩ = ⟨type⟩
```

In Section 5.3 we will see that Pascal has added a character data type for nonnumeric programming and a variety of data type constructors for arrays, records, sets, pointers, and so forth. Programmers can use these, in conjunction with type declarations, to design data types specifically suited to their applications.

Pascal's control structures incorporate many of the ideas of structured programming. Of course the **if-then-else** and **for**-loop (in a very simplified form) are provided. Pascal also provides leading and trailing decision loops and a **case**-statement for handling the breakdown of a problem into many cases. The **goto** is also provided.

5.3 DESIGN: DATA STRUCTURES

The Primitives Are Like Algol-60's

Recall that Algol-60 has three primitive data types: reals, integers, and Booleans. These, in turn, were very similar to the primitive data types provided by FORTRAN. This reflects the fact that both of these languages are predominantly *scientific* programming languages, and numbers and logical values are the most useful objects for scientific programming. Pascal extends its applicability to commercial and systems programming by providing one additional primitive data type, *characters.* A variable of type **char** (character) can hold exactly one character; longer strings of characters are manipulated as arrays of characters.

Enumeration Types Improve Security and Efficiency

Often programs must manipulate nonnumeric data; this is usually character data, but it can also be more abstract. For example, a commercial data-processing program may need to be able to deal with days of the week. In a language like FORTRAN or Algol, these would usually be encoded as integers, say in the range 0–6. This coding can make a program very hard to read unless it is carefully commented:

```
today := 2;               comment Tuesday;
tomorrow := today + 1;    comment advance to next day;
```

There is always the danger that the programmer will write the wrong code (do we start from Sunday or Monday?) or that the reader will misinterpret the code. There ought to be some way to avoid the error-prone use of numeric codes.

One approach that can be used in most languages is to initialize meaningfully named variables to the proper values. In Algol we could write

```
begin integer today, tomorrow;
   integer Sun, Mon, Tue, Wed, Thu, Fri, Sat;
   Sun := 0, Mon := 1; Tue := 2; Wed := 3;
   Thu := 4; Fri := 5; Sat := 6;
       .
       .
       .
   today := Tue;
   tomorrow := today + 1;
```

This is certainly more readable and less error-prone to write. It still has problems, however.

The major problem is that `today` and `tomorrow` are just integer variables and, as far as the compiler is concerned, it is meaningful to treat them as integers. For example, all of these statements are legal, although they make no sense on days of the week:

```
today := 355;
tomorrow := -3;
today := (tomorrow - 100)/today + 5280;
```

This is a lack of *security*; the compiler is allowing us to write statements that have no meaning (in our application domain).

The situation is even worse since we may have other logically different types that are represented as integers:

```
begin integer gender, ThisMonth;
   integer Jan, Feb, Mar, ... , Dec;
   integer male, female;
   Jan := 0; Feb := 1; ... ; Dec := 11;
```

```
  male := 0; female := 1;
   .
   .
   .
  today := male;
  gender := (female + Thu)/Dec;
  ThisMonth := 15;
```

Of course, errors as blatant as these are not likely to occur, but they illustrate the problem. There is a loss of transparency in the program and the mechanism intrudes.

To eliminate this lack of security, Pascal provides *enumeration types.* This is a mechanism for constructing types by *enumerating*, or listing, all their possible values. For example, we can declare the types for months, days of the week, and sexes like this:

```
type
  month = (Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  DayOfWeek = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  sex = (male, female);
```

(Notice that an enumeration type declaration is a *binding construct* since it binds the enumerated names, Sun, Mon, etc., to the values of the type.)

After these type declarations, it is possible to declare variables of these types and use them as usual:

```
var
  today, tomorrow: DayOfWeek;
  ThisMonth: month;
  gender: sex;
begin
    .
    .
    .
  today :=  Tue;
  today := tomorrow;
  ThisMonth := Apr;
  gender := female;
```

The security of these declarations is preserved by not allowing incompatible assignments (e.g., ThisMonth := male).

Pascal also preserves security by preventing the programmer from performing meaningless operations on enumeration values. Remember that there are two parts to an *abstract data type:* the set of data values and the primitive operations on those data values. For an enumeration type, the set of data values is specified in the enumeration. The operations do not have to be specified because they are the same for all enumeration types:

```
:=, succ, pred, =, < >, <, >, <=, >=
```

The *ordering relations* (<, >, etc.) are defined according to the order specified in the declaration of the enumeration type; for example, Mon < Wed and Dec > Jan. The **succ** and

pred (successor, predecessor) functions give the succeeding and preceding elements in the list; for example, **succ**(Mon) = Tue and **pred**(Mar) = Feb. These operations are also *secure*; for example, **succ**(Sat) and **pred**(Jan) are errors.

The implementation of enumeration types is very efficient. The compiler does essentially what programmers would have to do if they did not have enumeration types: It assigns consecutive integer codes to the values. There is an important difference (aside from security), however. When the programmer declares a variable of type `DayOfWeek`, the compiler knows that the only possible values are `Sun`, `Mon`, `...`, `Fri`. Therefore, the only values that can be assigned to this variable are integers in the range 0–6, seven different values. Since seven values can be represented in 3 bits, the compiler can use 3 bits to represent a `DayOfWeek` variable rather than the 16 or 32 that are usually required for an integer:

```
Sun  Mon  Tue  Wed  Thu  Fri  Sat
000  001  010  011  100  101  110
```

Operations such as **succ** and **pred** and the relations are very efficient since these are just implemented as simple integer operations.

We can summarize the benefits of enumeration types:

1. They are *high level* and *application oriented*; they allow programmers to say what they mean.
2. They are *efficient* since they allow the compiler to economize on storage, and the operations can be performed quickly.
3. They are *secure* since the compiler ensures that programmers cannot do meaningless operations.

■ ***Exercise 5-1:*** Write an enumeration type declaration for automobile manufacturers, such as might be used in an automobile registration database system.

■ ***Exercise 5-2:*** Write an enumeration type declaration for the different types of access that a particular user might have to a file (read, write, etc.).

■ ***Exercise 5-3:*** Write an enumeration type declaration for the different graphics file formats that a graphics file viewer might support.

Subrange Types Improve Security and Efficiency

We have seen that the enumeration type improves *security* since the compiler can check if the programmer is doing something meaningless, such as asking for the successor of the last element in the enumeration. The Pascal *subrange type constructor* extends this checking to integers and allows tighter checking on other types. Suppose the variable `DayOfMonth` is going to hold the number representing a day of the month; the meaningful values of this variable are the numbers 1–31. Although this could be declared as an integer variable, our program will be more secure if we use a subrange type:

```
var DayOfMonth: 1 .. 31;
```

This means that `DayOfMonth` can hold values in the range 1–31 (inclusive).

If we attempt to assign to this variable a value outside this range, we will get an error. It has been observed that if a programmer consistently uses subranges, then many program errors will manifest themselves as subrange violations. It is usually easier to find the cause of a subrange violation than the cause of the more subtle errors that often result if the violation is not caught.

Subrange declarations also allow the compiler to economize on storage utilization. For example, since `DayOfMonth` can take on only 31 different values, it can be stored in 5 bits (since $2^5 = 32$) rather than the 16 or 32 bits required for integers. This could make a big difference if we had a large array of such values.

Subrange types can be based on types other than integers. For example, we can declare a type `WeekDay` whose only possible values are the days Monday through Friday:

```
type WeekDay = Mon .. Fri;
```

(Note that we are assuming the previous definition of `DayOfWeek` so that the names `Mon` and `Fri` are defined.) Then, if we accidently assigned `Sat` or `Sun` to a variable of type `WeekDay`, we would get an error.

Pascal permits the programmer to define subranges of any *discrete type*, that is, enumeration types, integers, and characters. It does not permit defining a subrange of the real numbers, which is a *continuous type*.

▪ ***Exercise 5-4:*** Write a subrange type declaration for the type `age` representing the age of a child in school.

▪ ***Exercise 5-5:*** Write a subrange type declaration for the type `WaterTemp` representing water temperature in degrees Fahrenheit.

Set Types Are High Level and Efficient

Pascal provides the ability to manipulate small finite sets using the standard operations of set theory. As we will see, the set type is almost an ideal data type; it is high level and application oriented and yet very efficient. First, let's investigate its capabilities.

The description of a set type has the form

set of ⟨ordinal type⟩

where an ⟨ordinal type⟩ is an enumeration type (including **char** and **Boolean**), a subrange type, or a name of one of these. These restrictions on the *base type* of set types allow its efficient implementation.

For an example, suppose we have two variables of a set type:

```
var S, T: set of 1 .. 10;
```

This says that each of `S` and `T` is a variable that can hold a set of the numbers from 1 to 10. If we had declared `S` and `T` by

```
var S, T: 1 .. 10;
```

then we would have said that S and T can each hold exactly one number in the range from 1 to 10. With the set declaration, S and T can each hold any group (including zero) of numbers in this range.

How do we get a set into these variables? In mathematics, if we want to say that S is the set containing the numbers 1, 2, 3, 5, 7, we write

$$S = \{1, 2, 3, 5, 7\}$$

In Pascal, if we want to assign this set to the variable S, we write

```
S := [1, 2, 3, 5, 7]
```

We can see that Pascal has followed mathematical notation closely, although limited character sets have forced some deviations.

The sets in Pascal can be manipulated just like mathematical sets. For example, if we have assigned

```
T := [1 .. 6]
```

then T contains the set [1, 2, 3, 4, 5, 6]. If we then form the intersections $S \cap T$ and store it in T by

```
T := S * T
```

the result will be that T contains [1, 2, 3, 5]. Notice that the produce notation '*' is used instead of the usual set notation '$\cap$'. After the above operation is performed, T has the value [1, 2, 3, 5], which we can test with the equality relation:

```
if T = [1, 2, 3, 5] then ...
```

Similarly, the union (+) and difference (−) operations can be performed on sets.

If x is an integer in the proper range (1–10 in this case), then we can ask if x is in a set by writing x **in** S. This is just the set theory $x \in S$. For example, 3 **in** S is true, but 6 **in** S is false. Another common relation between sets is the *subset* relation. We say that $S \subseteq T$ if every member of S is also a member of T. In Pascal this is written S < = T. For example, if

```
S = [1, 2, 3, 5, 7], and
T = [1, 2, 3, 5]
```

then T < = S returns **true.** The relational operators '=', '< >' (not equal), '< =', and '> =' are provided among sets. Inexplicably, the proper set inclusion operations '<' and '>' are *not* provided. However, they are easy to implement; for example, S < T is equivalent to S <= T **and** S <> T.

■ ***Exercise 5-6:*** Show that the preceding definition of proper set inclusion (<) is correct.

The set-type constructor is very high level; it allows the programmer to write algorithms in a natural notation. For example, suppose that we are writing a compiler; it is necessary to categorize the characters in the program we are reading. This can be done by defining sets of characters that reflect the different categories, as is shown in Figure 5.2. Once the sets `digits`, `letters`, and `punctuation` have been defined, any character can be classified by testing to see which set it is in. To find out if the character in `ch` is alphanumeric, we just write

```
ch in letters + digits
```

since the alphanumeric characters are just the union of the letters and the digits.

One of the best characteristics of the set-type constructor is its efficiency. Consider our example, `S`, which is of type **set of** `1 .. 10`. This means that there are 10 potential members of any set stored in `S`, the numbers 1–10. To know what set is stored in `S`, it is only necessary to know, for each of these 10 numbers, whether that number is or is not in the set. This means that the set `S` can be represented by 10 bits, with each bit being one if the corresponding value is in the set and zero if it is not. For example, the set `[1, 2, 3, 5, 7]` is represented by the bits

```
S  =  1  1  1  0  1  0  1  0  0  0
      ↑  ↑  ↑  ↑  ↑  ↑  ↑  ↑  ↑  ↑
      1  2  3  4  5  6  7  8  9  10
```

This means that 1 is in the set, 2 is in, 3 is in, 4 is not, 5 is, and so on. This is certainly a compact representation: We have represented a set containing up to 10 numbers in only 10 bits! Other sets require different size bit strings. For example, a **set of** `WeekDay` will

```
var digits, letters, punctuation: set of char;
  ch: char;
begin
  digits := ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];
  letters := ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
             'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
             's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
             'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
             'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
             'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'];
  punctuation := ['(', ')', '.', ',', ';', ':', '!'];
     .
     .
     .
  read(ch);
  if ch in letters then ... handle letters ...
  else if ch in digits then ... handle digits ...
  else if ch in punctuation then ... handle punctuation ...
  else ... handle illegal characters ...
     .
     .
     .
end
```

Figure 5.2 Using Sets to Classify Characters

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

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

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

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

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

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

The other relations are implemented similarly.

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

The Elegance Principle

Confine your attention to designs that *look* good because they *are* good.

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

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

Array Types

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

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

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

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

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

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

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

Mon	Hours
Tue	
Wed	
Thu	
Fri	

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Problems with Array Bounds

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The syntax of a simple conformant array schema is

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

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

- ***Exercise 5-9*:*** Discuss conformant array schemas as a solution to the array parameter problem. Do you think the right decision was made? Can you suggest an alternative?

- ***Exercise 5-10*:*** Suppose strings are represented as arrays of characters. What problems would you face upon implementing a string manipulation package in Revised Report Pascal (i.e., Pascal without conformant array schemas)? Discuss possible solutions.

- ***Exercise 5-11*:*** Write the procedure headers for a string manipulation package for ISO Standard Pascal (i.e., Pascal with conformant array schemas).

- ***Exercise 5-12:*** Write a set of Pascal procedures for performing mathematical operations (sum, inner product, length, etc.) on real vectors of any dimension.

- ***Exercise 5-13*:*** The ISO Pascal Standard defines two "levels of compliance." A compiler complies at level 1 if it implements all of the standard *except* conformant array schemas; it complies at level 2 if it implements all of the standard *including* conformant array schemas. Why do you suppose there are these two levels of compliance? Discuss the pros and cons.

Record Types Group Heterogeneous Data

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

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

```
type person =
  record
    name: string;
    age: 16 .. 100;
    salary: 10000 .. 100000;
    sex: (male, female);
    birthdate: date;
    hiredate: date;
  end;

string = packed array [1 .. 30] of char;

date = record
    mon: month;
    day: 1 .. 31;
    year: 1900 .. 2100;
  end;

month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
         Nov, Dec);
```

Figure 5.3 Example of a Record Type—A Personnel Record

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

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

```
var newhire: person;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Composite Data Structures

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

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

Variant Records Allow Alternative Structures

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

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

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

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

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

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

Figure 5.4 Record without Variants

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

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

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

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

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

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

```
type plane = record
  flight:          0 .. 999;
  kind:            (B727, B737, B747);

  case status:     (inAir, onGround, atTerminal) of
  inAir: (
    altitude:      0 .. 100000;
    heading:       0 .. 359;
    arrival:       time;
    destination:   airport);

  onGround: (
    location:      airport;
    runway:        runwayNumber);

  atTerminal: (
    parked:        airport;
    gate:          1 .. 100;
    departure:     time)

end {plane};
```

Figure 5.5 Type Declaration for Record with Variants

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

Pascal's Data Structures Exhibit Responsible Design

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

The Responsible Design Principle

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

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

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

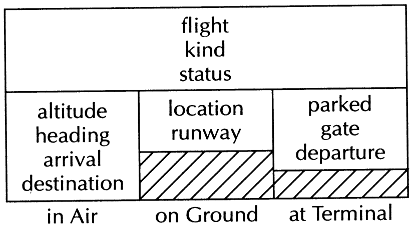

Figure 5.6 Memory Layout of a Variant Record

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

Pointer Types Are Secure

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

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

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

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

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

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

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

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

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

↑ ⟨type name⟩

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

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

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

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

```
var p:  ↑plane;
```

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

```
p↑.parked[1]
```

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

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

Type Equivalence Was Not Clearly Specified

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

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

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

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

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

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

This last example suggests the problem with structural equivalence: If programmers declare two types, called `person` and `car`, then they probably intend them to represent different things, and it probably does not make any sense to assign one to the other. The fact that they happen to be defined by the same record structure may be a coincidence. This leads to another rule for type equivalence, called *name equivalence.* The Name Equivalence Rule says that two objects have the same type if the *names* of their types are the same. Therefore, in the last example, `x := y` is illegal since the type of `x` is `person` and the type of `y` is `car`, and these are two different names. What about the previous example?

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

Different versions of the Name Equivalence Rule differ on whether `x` and `y` have the same type or not. One common interpretation says that they have different types since the compiler makes up different names for these *anonymous types*; it is as though the program were

```
type  T00029 = record id: integer; weight: real end;
      T00030 = record id: integer; weight: real end;
var   x: T00029;
      y: T00030;
```

Name equivalence has some problems of its own. Consider this example:

```
type  age = 0 .. 150;
var   n: integer;
      a: age;
```

Is the assignment `n := a` legal? Pure name equivalence says "no" since `n` and `a` have different **type** names (namely, **integer** and `age`). In this case the Revised Pascal Report explicitly allows this assignment since it says that two objects are considered to have the same type if the type of one is a subrange of the type of the other. This is an exception to the general Name Equivalence Rule and the programmer must remember this exception.

Which is better, structural equivalence or name equivalence? There has been a lot of debate on this among programming language designers. The general consensus seems to be that name equivalence is better. There are two reasons:

1. Name equivalence is generally safer since it is more restrictive. It is presumed that if programmers declare the type twice, then they probably have a reason for doing it—the two types probably represent different things. Therefore, considering these types different protects the *security* of the program (it prevents the programmer from doing something meaningless).

2. Name equivalence is simpler to implement; the compiler has to compare only the character strings representing the names of the types. For structural equivalence, it is necessary to write a recursive function that compares the data structures representing the types of the two objects. This also slows down the compiler.

It is for reasons such as these that the ISO Pascal Standard specifies that name equivalence be used, although it deviates from pure name equivalence in several ways. We will see in Chapter 7 that Ada has also adopted a modified form of name equivalence that attempts to provide the security of name equivalence with the flexibility of structural equivalence.

■ ***Exercise 5-16*:*** Can you think of any circumstances under which the Structural Equivalence Rule is preferable? Suppose you have a function that returned the sum of the elements of a `1 .. 100` array of reals. Wouldn't it be convenient if this function could be used on any such array type, no matter what it is called? Discuss.

5.4 DESIGN: NAME STRUCTURES

There Are Six Name-Binding Mechanisms

Pascal provides six kinds of primitive name structures, or methods of binding names to their meanings:

1. Constant bindings
2. Type bindings
3. Variable bindings
4. Procedure and function bindings
5. Implicit enumeration bindings
6. Label bindings

The fifth of these we discussed in Section 5.3, where we discussed enumeration types. There is little to say about label bindings, beyond the fact that all statement labels must be listed in the declaration part of the procedure containing the statement they label. The other binding mechanisms are discussed in this section.

Constants Aid Readability and Maintainability

Pascal includes the ability to name *constants* of discrete types (integers, enumerations, and characters). Constant declarations are introduced by the word **const** and have the syntax

⟨name⟩ = ⟨constant⟩;

What is the importance of constant declarations?

Consider a large program that manipulates arrays. In such a program, there will normally be many dependencies between the dimensions of these arrays and other parts of the program. For example, if the arrays have dimensions `[1..100]`, then there will normally be **for**-loops with bounds `1 to 100` or `100 downto 1`. Further, there may be other arrays that have dimensions `[1..100]`, `[0..99]`, and so forth, in order to be compatible with the first array. All of these interdependencies are *implicit*, although proper documentation can help to make them explicit.

Next, suppose that in the normal process of maintaining this program it is decided that the size of one of these arrays should be changed to `[1..150]` (say, to accommodate larger data sets). This will imply that the dimensions of some of the other arrays also be changed and that the limits on some of the **for**-loops be changed. The programmer will have to find all numbers that *implicitly depend* on the changed dimensions and make appropriate changes to these numbers. It is not as simple as using an editor to change all '`100`'s to '`150`'s since some '`100`'s depend on the array's dimensions and some don't. Rather, each instance of '`100`' must be considered individually. This is a very error-prone and tedious process.

Pascal's constant declarations solve many of these problems since they allow implicit dependencies to be made explicit. They do this through the application of the Abstraction Principle since all the dependent constants are abstracted out of the program and are given a name. For example, given the declaration

```
const MaxData = 100;
```

the programmer can use the named constant `MaxData` in all declarations, **for**-loops, and so on, that depend on this parameter. (Instances of '`100`' that do not depend on it would presumably have their own names.) Changing this parameter to `150` is then simple since only the constant declaration has to be changed; all other dependent changes are made automatically.

We must mention important limitations of Pascal's constant declarations: The constant cannot be described by an expression, and expressions cannot be used in variable and type declarations. For example, suppose that a dependent array `A` had dimensions `[0..99]`; we should declare this as

```
var A: array [0 .. MaxData - 1] of real;
```

Unfortunately, this is not allowed since Pascal does not permit expressions (even constant expressions like `MaxData - 1`) to be used as array dimensions or subrange bounds. The best we can do is to declare two constants and document the dependency:

```
const  MaxData = 100;
       MaxDataMinus1 = 99; { = MaxData - 1 }
var    A: array [0 .. MaxDataMinus1] of real;
```

Some newer languages (e.g., Ada, Chapter 7) have eliminated this restriction.

The Constructor Is a Simplification of Algol's

There are two major name structure constructors in Pascal. One is the record-type constructor, which is discussed in Section 5.3. The other constructor is the procedure (or function), which is the major scope defining construct.

A procedure declaration has the form

```
procedure ⟨name⟩ (⟨formals⟩);
  ⟨declarations⟩
begin
  ⟨statements⟩
end;
```

This is very similar to the Algol procedure declaration:

```
procedure ⟨name⟩ (⟨formals⟩); ⟨formal specifications⟩
begin
  ⟨declarations⟩
  ⟨statements⟩
end
```

We can see that the main difference is that the parts have been rearranged a little.

The scope rules of Algol and Pascal bindings are essentially the same. The scope of the ⟨declarations⟩ is the entire block, including all the ⟨declarations⟩ and the ⟨statements⟩. The scope of the ⟨formals⟩ includes both the local ⟨declarations⟩ and the ⟨statements⟩ in the body of the procedure.

The ISO Pascal Standard places an additional restriction on the order of declarations to permit one-pass compilation. The scope of each of the ⟨declarations⟩ includes that declaration, all of the following declarations, and the ⟨statements⟩. This means that names must be bound before (textually) they are used, whereas in Algol and Revised Report Pascal this is not the case. This restriction simplifies the compiler since it means that declarations can be processed in a single pass across the source. However, there are two problems.

First, structured programming methods encourage programmers to structure their programs in a *top-down* order. That is, the uppermost procedures are defined first, then the lower level ones that they call, and then the lower level ones that these call, and so on. This is *exactly the opposite* order from that required by ISO Standard Pascal, since it means that every procedure will be called before (textually) it is defined.

The second problem with this restriction is that mutually recursive procedures *cannot* be defined before they are called. For example, suppose that P calls Q and Q calls P. The following declarations are incorrect since Q is used before it's declared:

```
procedure P ( ... );
begin
        .
        .
        .
    Q ( ... );
        .
        .
        .
end;
procedure Q ( ... );
begin
        .
        .
        .
    P ( ... );
        .
        .
        .
end;
```

To solve this problem, ISO Pascal provides a **forward** declaration, which gives the compiler the information it needs before the procedure is actually declared. For example,

```
procedure Q ( ... ); forward;
procedure P ( ... );
begin
        .
        .
        .
    Q ( ... );
        .
        .
        .
end;
procedure Q;
```

```
begin
      .
      .
      .
   P ( ... );
      .
      .
      .
end;
```

Most Pascal dialects (including the ISO Standard) require the declarations in a procedure or program to be in a particular order: labels, constants, types, variables, and subprograms:

```
procedure ⟨name⟩ (⟨formals⟩);
  ⟨label declarations⟩
  ⟨const declarations⟩
  ⟨type declarations⟩
  ⟨var declarations⟩
  ⟨procedure and function declarations⟩
begin
  ⟨statements⟩
end
```

This is not an unreasonable order for declarations to be in. As we can see in Figure 5.1, constant declarations are often used in succeeding type declarations; type declarations are used in succeeding variable declarations; and the variables are used in the subprograms. Requiring this order can be inconvenient in large programs, however, since it prevents the user from grouping together related constant, type, variable, and subprogram declarations. This problem is solved by the encapsulation mechanisms provided in fourth-generation languages (Chapter 7).

Pascal Eliminates the Block

Recall the distinction in Algol-60 between a *block* and a *compound statement*: A block contains local declarations and statements, whereas a compound statement contains only statements. Blocks facilitate the sharing of storage since arrays declared in disjoint blocks can occupy the same memory locations. This is not possible in Pascal since Pascal provides compound statements but no blocks. In Pascal, storage can be shared only between disjoint *procedures*. Although this simplifies Pascal's name structures, it complicates efficient use of memory.

▩ ***Exercise 5-17:*** Recall the example on p. 112 of storage sharing in Algol-60. How would you program this in Pascal so that the arrays x and m share storage?

5.5 DESIGN: CONTROL STRUCTURES

Control Structures Reflect Structured Programming Ideas

Pascal has more control structures than Algol-60, although they are simpler than Algol-60's. As in Algol-60, the control-structure constructors route control among the basic computational primitives of the language: those operations that fetch, store, and operate on the values of variables. In addition, Pascal provides simple input-output statements (missing from Algol-60), since it was designed for teaching.

In Chapter 3 (Section 3.5), we described how Algol's ability to nest control structures led to the ideas of structured programming. Pascal continues this development by providing several more structured control structures. Each of these is characterized by having one entry point from the previous statement and one exit point to the following statement. This is exemplified by Pascal's **if-then-else** statement, which is exactly like Algol's. This single entry–single exit property simplifies programs by satisfying the Structure Principle: The static structure of the program corresponds in a simple way to its dynamic structure.

Pascal does have a **goto**-statement, although the richness of the other control structures means it is rarely needed. As in Algol, nonlocal transfers are permitted. At the time Pascal was designed, Wirth thought that the absence of a **goto** would discourage too many people from using the language; he did eliminate the **goto** from the next language he designed, Modula.

Pascal, like almost all languages designed since Algol, has fully recursive procedures.

The for-Loop Is Austere

Recall the complexity of Algol's **for**-loop; it included stepping elements, **while**-conditions, and lists of arbitrary values. Furthermore, the expressions in the bounds were evaluated on every iteration, giving them the ability to change from one iteration to the next. This is exactly the kind of baroque, inefficient control structure that Pascal tries to avoid. As we will see, Pascal's **for**-loop is even simpler than FORTRAN's DO-loop!

The syntax for a Pascal **for**-loop is

$$\textbf{for}\ \langle\text{name}\rangle\ :=\ \langle\text{expression}\rangle \begin{Bmatrix}\textbf{to}\\ \textbf{downto}\end{Bmatrix} \langle\text{expression}\rangle\ \textbf{do}\ \langle\text{statement}\rangle$$

The loop can step up by + 1 increments or down by − 1 increments, as indicated by the words **to** and **downto;** step sizes other than 1 are not allowed. Critics have maintained that this is carrying things too far because nonunit step sizes are very useful and do not add much to the complexity of the language. This is one case where Pascal may have overreacted to the complexity of the second-generation languages.

Pascal also specifies that the bounds of the loop are computed once, at loop entry, rather than on every iteration as is done in Algol. The result is that a Pascal **for**-loop always executes a definite number of times (provided, of course, the programmer does not jump out of it with a **goto**). For this reason the Pascal **for**-loop is both more efficient and easier to understand. It is an example of a *definite iterator*, that is, a structure that iterates a definite (predictable) number of times. We will see below that Pascal also provides indefinite iterators.

■ ***Exercise 5-18:*** Write a Pascal **for**-loop to sum into S the elements of

```
var A: array [min .. max] of real;
```

■ ***Exercise 5-19:*** How would you write a Pascal **for**-loop to sum the even-indexed elements of the array declared:

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

▨ ***Exercise 5-20*:*** Discuss Pascal's **for**-loop. Is it too simple? Should a variable step size be provided? Suggest a syntax and justify its inclusion. Should it be even simpler (e.g., not allow **downto**)? Justify this choice. Should it be eliminated altogether since **for**-loops can be implemented using the indefinite iterators?

▨ ***Exercise 5-21*:*** The ISO Pascal Standard requires the controlled variable of a **for**-loop to be a *local variable* of the procedure or function containing the loop. Why have the language designers imposed this restriction? Discuss the trade-offs pro and con.

Leading- and Trailing-Decision Indefinite Iterators

Pascal provides two constructs for *indefinite iteration*, that is, for looping when the exact number of iterations is not known at loop entry. In these cases, a *condition* that's tested on each iteration determines whether or not the loop has completed.

In Chapter 2 (Section 2.3), we discussed two places in the loop where the decision can be placed—the beginning and the end of the loop. These two situations are handled by Pascal's **while**-loop and **repeat**-loop, respectively:

```
while ⟨condition⟩ do ⟨statement⟩
repeat ⟨statements⟩ until ⟨condition⟩
```

The **while**-loop is an example of a *leading-decision* iterator and the **repeat**-loop is an example of a *trailing-decision* iterator.

Although these two constructs handle most loops, there are other places the decision can be put. For example, a *mid-decision* iterator places the decision in the middle of the loop. For these Pascal must use a **goto** and an impossible termination condition (**while true**):

```
while true do
  begin
     . . . first half of loop body . . .
    if ⟨done⟩ then goto 99;
     . . . second half of loop body . . .
  end;
99:
```

Some newer languages (e.g., Ada, Chapter 8) have provided a mid-decision indefinite iterator.

▨ ***Exercise 5-22:*** Suggest a situation in which a mid-decision loop would be useful. *Hint:* Recall the example of an Algol-60 **for-while** loop in Section 3.5.

▨ ***Exercise 5-23*:*** Do you think Pascal should include a mid-decision loop? Discuss the relative merits of a **goto** and a mid-decision loop. Can the **goto** be eliminated if a mid-decision loop is included? Should it be eliminated? Can all of the indefinite iterators be handled by one loop statement? Suggest a possible syntax.

The case-Statement Is an Important Contribution

It is quite common in programming (and problem solving in general) to break a problem down into two or more subproblems. That is, a problem is often divided into several *cases* that are handled differently. For example, if we are processing `plane` records (Section 5.3), then we may want to divide them into three cases, depending on whether their `status` is `inAir`, `onGround`, or `atTerminal`. This is handled by the Pascal **case**-statement.

Other languages also have mechanisms for handling cases; for example, FORTRAN provides the computed GOTO. If we want to do one group of statements (S_1) if `I` = 1, another (S_{23}) if `I` = 2 or 3, and a third (S_4) if `I` = 4, then we can write

```
      GOTO (100, 250, 250, 400), I
100     ... S1 ...
      GOTO 500
250     ... S23 ...
      GOTO 500
400     ... S4 ...
500     ... rest of program ...
```

This is quite efficient since the computed GOTO is compiled as a jump table; but it is not very readable since there is no way to tell from the GOTO where the labels are. They could be anywhere in the subprogram so the flow of control is not obvious. This violates the Structure Principle.

Several programming languages have attempted to provide more structured alternates to FORTRAN's computed GOTO and Algol's **switch.** These were patterned on the **if-then-else,** which breaks the problem into two cases depending on the condition. These constructs are called **case**-statements and usually look something like the following:

```
case ⟨expression⟩ of
  ⟨statement⟩,
  ⟨statement⟩,
       ⋮
  ⟨statement⟩
end case;
```

The meaning is as follows: If the value of ⟨expression⟩ is 1, the first ⟨statement⟩ is executed; if it is 2, the second is executed, and so forth. Like the **if-then-else,** the **case**-statement has a single entry from the preceding statement and a single exit to the following statement.

There are several problems with this construct. First, if there are many cases (and **case**-statements with 50 or 100 cases are not unusual), then it may be difficult for the reader to tell which statements correspond to which cases. Commenting on the statements with the case numbers helps, but there is still the problem of incorrect comments: programmers often change the code without bothering to change the comments.

Another problem is that this construct works only for cases that are indexed by integers beginning with 1. This kind of **case**-statement will not handle other data types, such as `(inAir, onGround, atTerminal)`, as the basis for the case breakdown.

Another problem is that it is difficult to handle two or more cases with the same code. For our previous example, where both cases 2 and 3 were handled by the code S_{23}, we would have to write

```
case I of
   begin ... S1 ... end,
   begin ... S23 ... end,
   begin ... S23 ... end,
   begin ... S4 ... end
end case
```

Notice that we have repeated the statements S_{23} for each of the cases 2 and 3. This violates the Abstraction Principle and makes the program harder to read, harder to write, and harder to maintain. In particular, since it is not obvious that the code for cases 2 and 3 is the same, one might fix a bug in one without realizing that it also has to be fixed in the other. We could factor out the code S_{23} into a procedure `P` and call this procedure from each of cases 2 and 3, but this approach tends to clutter the program with procedures that are not very meaningful.

A solution to all of these problems can be found in Pascal's *labeled* **case**-*statement* designed by C. A. R. Hoare. Hoare has said that this is the most important of his many contributions to language design.

Like the unlabeled **case**-statement described above, the labeled **case**-statement contains one or more case clauses:

```
case ⟨expression⟩ of
  ⟨case clause⟩;
  ⟨case clause⟩;
        ⋮
  ⟨case clause⟩
end
```

Unlike in the unlabeled **case**-statement, however, each case clause begins with one or more constants that *label* that clause:

```
⟨constant⟩, ⟨constant⟩ ,  ... : ⟨statement⟩
```

This solves the problem of handling cases 2 and 3 with the same code:

```
case I of
   1:        begin ... S1 ... end;
   2, 3:     begin ... S23 ... end;
   4:        begin ... S4 ... end
end
```

Notice that this is more *secure* than the unlabeled **case**-statement; there is no possibility that the case label is incorrect as there was when they were just comments. The cases can even be out of order. The programmer can be assured that the case labeled 1 will be executed when `I` is 1. Another way of looking at this is that the labeled **case**-statement is *self-*

documenting; that is, it does not depend on the programmer properly documenting which case is which. The labeled **case**-statement illustrates the Labeling Principle.

The Labeling Principle

Avoid arbitrary sequences more than few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.

The labeled **case**-statement is also more flexible than the unlabeled **case**-statement. In particular, values other than integers can be used to discriminate between the cases. Our `plane` example can be written

```
case NextFlight.status of
   inAir:                 . . . handle plane in air  . . . ;
   onGround:              . . . handle plane on runway  . . . ;
   atTerminal:            . . . handle parked plane  . . .
end
```

This saves the programmer from the error-prone process of manually mapping plane `status` into integers and provides clear documentation of the case breakdown.

The **case**-statement is also quite efficient. Since the compiler can determine the type of the case-selection expression, it knows the possible values that this expression can have. It can then construct a jump table of the appropriate size for the **case**-statements. Some compilers go even further and generate either a jump table, a hash table, a binary search of a sorted table, a sequential search of a linear table, or a series of **if**-tests, depending on the number of values, their range, and their clustering.

In summary, the labeled **case**-statement is just as efficient as the computed `GOTO`, but it is higher level, more secure, more readable, and more structured.

■ ***Exercise 5-24:*** Write a **case**-statement that assigns to a variable `days` (of type `28 .. 31`) the number of days in the month stored in `ThisMonth` (of type `month`).

■ ***Exercise 5-25*:*** Consider at least three ways of implementing a **case**-statement (e.g., sequential test, jump table, and binary search). Assuming all the cases are equally likely, derive formulas for the average amount of time to execute a **case** given each of the implementation methods. State parameters by which a compiler could decide which implementation to use for any give **case**-statement.

■ ***Exercise 5-26*:*** Perform an analysis similar to that in the previous exercise, but compute the average amount of storage required for the **case.** Suggest ways of trading off space against time.

Parameters Are Passed by Value or Reference

The Revised Report on Pascal describes two parameter passing modes: pass by value and pass by reference. Pass by reference replaces the expensive and confusing *name* parameters of Algol-60 with the simpler mechanism provided by FORTRAN. The purpose of pass by reference is to allow a procedure to alter its actual parameters; in other words, reference parameters are output (or input-output) parameters. The problem with pass by reference in FORTRAN was that a procedure could alter actuals that it made no sense to alter, such as literal constants (see Chapter 2, Section 2.3). Pascal solves this problem since the programmer must specify in the procedure declaration whether a parameter is to be passed by value or reference. If a parameter is passed by reference, then the Pascal compiler will ensure that the corresponding actual is something that it is meaningful to store into, that is, a variable, array element, or record field. Therefore, in Pascal, reference parameters obey the Security Principle.

Reference parameters are also *efficient* since only an address is passed, no matter how large the corresponding actual is. In contrast, name parameters are expensive since a *thunk* (Chapter 3, Section 3.5) must be invoked on each use of the parameter.

Pass by value in Pascal is exactly like pass by value in Algol; a copy of the actual parameter is made in a variable local to the procedure. This is intended for input parameters since it prevents modification of the actual. It has the same efficiency problems as Algol's pass by value since an array or other large data structure will have to be copied. Unfortunately this leads some programmers to pass by reference parameters that are not intended for output.

Pass as Constant Was Once Provided

The original Pascal Report did not specify pass by reference and *pass by value* as the parameter passing modes; rather, it specified pass by reference and *pass as constant.* Constant parameters are similar to value parameters in that they are intended for inputs, but they have an efficiency advantage that value parameters do not. We discuss this below.

A parameter passed by constant is considered a constant within the body of the procedure; therefore, it is not legal to use that parameter as the destination of an assignment. It is also illegal to use it in any other context where it is a destination, such as an actual parameter passed by reference. Thus, the compiler protects the security of constant parameters.

Since the compiler prevents a constant parameter from being altered, the compiler can either pass the value of the actual or pass its address, whichever is more efficient. For example, for small values, such as integers, characters, and elements of enumeration types, the compiler can copy the value of the actual parameter into local storage, as it would for pass by value. For large values, such as arrays and large records, the compiler can just pass the address of the actual. This saves the time and extra space required for a copy of the parameter with pass by value. There is no danger of the actual being inadvertently modified because the compiler prevents assignments to constant parameters. (In Chapter 8, Section 8.1, we analyze quantitatively the trade-offs between passing addresses or copies of parameters.)

We can see that constant parameters have the security advantages of value parameters and the efficiency advantages of reference parameters. So why were they eliminated from Pascal and replaced with value parameters in the Revised Report (and the ISO Standard)? One possibility is that the security is not airtight. Pascal permits limited forms of *aliasing*. For example, the same variable may be visible in two ways, as a nonlocal and as a parameter. Consider this program fragment

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

```
   procedure P (x: vector);
   begin
      writeln (x[1]);
      A[1] := 0;
      writeln (x[1])
   end;
begin
   P(A)
end.
```

Notice that within the procedure P the same area of storage has two names: x and A. Since x is a constant parameter, the compiler may choose to pass the address of A to P. Then, since the assignment to A alters the value of the array, the two references to x[1] may produce different values even though there is no assignment to x between them. The illusion is imperfect; the mirrors can be seen.

Unfortunately, the alternative chosen—providing only value and reference parameters—encourages programmers to pass things by reference for the sake of efficiency, even though they have no intention of altering the actual from the procedure. This undermines the security of their programs and misleads the reader.

The problem is that two *orthogonal* issues are being confused: (1) the issue of whether a parameter is to be used for input or output and (2) the issue of whether to copy its value or pass the address of its value. Newer languages such as Ada (Chapter 8) have separated these decisions better by returning to a mode similar to constant parameters.

▩ ***Exercise 5-27:*** Write a Pascal fragment that uses parameters, but not nonlocal variables, to illustrate the security loophole in parameters passed as constants.

Procedural Parameters Are a Security Loophole

Pascal allows *procedural* and *functional* parameters, that is, procedures and functions passed as arguments to other procedures and functions; this restores some of the flexibility lost by omitting name parameters. For example, suppose we need to define the function difsq (f, x), which computes $f(x^2) - f(-x^2)$ for any real function f and real number x. Thus,

$$\text{difsq (sin, } \theta) = \sin(\theta^2) - \sin(-\theta^2)$$

In the Revised Report Pascal dialect, difsq would be defined:

```
procedure difsq (function f: real; x: real): real;
   begin
      difsq := f(x*x) - f(-x*x)
   end;
```

This seems straightforward enough.

Unfortunately, this method of specifying a functional parameter introduces a serious security loophole into Pascal. Notice that the procedure header for difsq only specifies that f is a **real** function; it says nothing about the number or types of the parameters of f. This makes it almost impossible for the compiler to determine if a particular call of `difsq` is legal or not (recall that `difsq` could itself be passed to another procedure).

For this reason the ISO Pascal Standard requires the programmer to specify the arguments of a formal procedure parameter, for example,

```
procedure disfsq (function f(y: real): real; x: real): real;
  begin .... end;
```

This allows the compiler to do complete type checking, thereby preserving Pascal's strong typing. The correct implementation of `difsq` is shown in Figure 5.7.

There are several costs associated with the solution just discussed. One obvious cost is increased language complexity. A more subtle cost is the implication that procedures are first-class citizens. Since this solution allows procedure types to be used in formal parameter specifications, the user could also reasonably expect to use them in variable declarations. If this were allowed, then the language would have to define the meaning of procedure variables, arrays of procedures, procedure-valued functions, files of procedures, and so on. There are many complex language-design and implementation issues involved in these concepts (some of which are discussed in later chapters). On the other hand, if the language did not permit procedure types to be used in variable declarations, then procedure types would not be first-class citizens and the user would have to learn more exceptions. This also adds to the complexity of the language.

▩ ***Exercise 5-28:*** Discuss the ISO Standard's way of providing secure procedural and functional parameters in Pascal. Defend the ISO solution to this problem or propose and defend your own solution.

```
var theta, phi: real;

function sin (x: real): real;
  begin .... end;

function exp (x: real): real;
  begin .... end;

function difsq (function f (y: real): real; x: real): real;
  begin
    difsq := f(x*x) - f(-x*x)
  end;

begin
  readln (theta); readln (phi);
  writeln (difsq (sin, theta));
  writeln (difsq (exp, phi))
end.
```

Figure 5.7 Example of Function Passed as Parameter

5.6 EVALUATION AND EPILOG

Pascal Has Lived Up to Its Goals

Recall that Pascal's primary goal was to be a good language for *teaching* programming. This led to subsidiary goals of reliability, simplicity, and efficiency. Even in the face of some of the problems we have discussed, Pascal has been very successful in these areas. Many universities and colleges now teach Pascal as a first programming language. This use has been encouraged by the good Pascal implementations that exist on almost all computers, including microcomputers. The quality of these implementations must in part be attributed to the small size and careful design of the language. Wirth (1993) observes, "the principle to include features that were well understood, in particular by implementors, and to *leave out* those that were still untried and unimplemented, proved to be the most successful single guideline." The second most important principle was, he says, to publish the language definition after the implementation was complete, thus making it a description of work completed, not of work planned. In many respects Pascal illustrates the Elegance Principle.

Much of the criticism Pascal has received results from trying to use it for purposes for which it was not designed. For example, Pascal has been criticized for its lack of a separate compilation facility, even though such a facility is not especially important in teaching programming (the language's intended application). Indeed, it is to Pascal's credit that it has been so successfully applied in so many areas for which it was *not* intended.

Pascal Has Been Extended

Although Pascal was intended as a teaching language, many programmers have found that it is also suitable for "real" programming including systems implementation. Its strong typing simplifies debugging and helps catch latent errors in production programs; its rich set of efficient, high-level data types simplifies many nonnumerical programs; and its small size means that a programmer can acquire mastery of the language in a moderate amount of time.

These qualities have made Pascal an attractive vehicle for programming language research. Pascal has been extended for concurrent programming, to support verification, and for operating systems writing. Like Algol before it, Pascal has become a basis for almost all new language designs; most new languages are "Pascal-like." This includes the language Ada, which is the topic of Chapters 7 and 8. Other offspring include Concurrent Pascal, Mesa, and Euclid. Wirth himself has designed a successor to Pascal, called Modula-2, which attempts to correct some of Pascal's deficiencies and to be a practical tool for systems programming, and more recently he has designed Oberon.

Subsets of PL/I Were Designed for Systems Implementation

In the Algol lineage, we have seen how the third-generation language Pascal was motivated by an emphasis on simplicity and efficiency, which was a reaction to the overgenerality and complexity of second-generation languages, such as Algol-68. In other language families there was a similar reaction to the second generation, which did not always lead to third-generation languages. For example, attempts to apply PL/I to systems programming were hampered by its inefficiency and the practical unpredictability of the costs of some constructs (violations of the Localized Cost Principle). Therefore, language designers developed a number of subsets of PL/I, such as PL/S and XPL, intended for systems implementation. Since in many cases these were intended for systems implementation on specific computers, there was not much concern for machine independence, and many machine-dependent constructs were included for the sake of efficiency. (We can compare the case of FORTRAN II, which had many constructs specific to the IBM 704.) The extreme case of these "MOHOLs" (machine-oriented higher-order languages) were the various "structured assemblers," which provided direct access to a computer's instruction set, but allowed the use of high-level language syntax, including infix operators.

BCPL Was Designed for the Implementation of CPL

A similar evolutionary development can be seen in another language family. In the early 1960s teams at Cambridge and London Universities developed a semantically sophisticated but very complex language called CPL—explained as "Cambridge Plus London" or "Combined Programming Language." The latter acronym hints at its complexity, for CPL exhibits the full Baroque flowering of the second generation. Like Algol 60, CPL posed implementation challenges for its designers, and so they wanted to implement it in the best programming language: CPL. (This is not unusual—Pascal was implemented in Pascal; it was expected of any decent general purpose language that it would be the best vehicle for implementing its own compiler.) To simplify this process Martin Richards designed (1967) a subset of CPL, called BCPL for "Basic CPL," which included just those features essential for systems implementation. Ultimately, the CPL project died away, but BCPL became a moderately popular systems implementation language in the early 1970s.

B Was Designed for the Implementation of Unix

In particular BCPL was popular at Bell Labs when the earliest versions of Unix (for an 8K PDP-7!) were being developed. Since the Unix project needed a systems implementation language, in 1969–1970 Ken Thompson designed a language called "B"; "it is BCPL squeezed into 8K bytes of memory and filtered through Thompson's brain."[6] At first there was little concern for portability, and the language was very close to the machine. In particular, like BCPL and many other systems implementation languages, B was typeless; that is, it had a single data type corresponding to a word of PDP-7 memory. This is, of course, the extreme of weak typing, and is more typical of an assembly language than of even first-generation languages, which typically have several data types and some notion of type checking. Many other language design decisions were dictated by the limited memory available to compile B on the PDP-7.

[6] The quotations and historical information on C are from Dennis M. Ritchie, "The Development of the C Language" [*SIGPLAN Notices 28*, 3 (March 1993), pp. 201–208].

C Evolved Gradually from B

With the arrival of a PDP-11 in 1970, the Unix team became aware of problems with B's addressing scheme, which was incompatible with memory addressing on the PDP-11. Therefore, in 1971 Dennis M. Ritchie began extending B to include rudimentary data types (for purposes of memory allocation and addressing, not type checking), inspired in many respects by Algol-68. Eventually he called this language "C," as the successor of "B"; its evolution was mostly complete by 1973, when it was used to rewrite the kernel of the Unix operating system.

Some additional third-generation features, such as union and enumeration types, were added in the late 1970s, but attempts to port Unix to other computers accented the portability and security problems of weak typing. Therefore a more restricted type system was designed, but it was not enforced by most compilers; instead programmers had to rely on a separate type-checker (called `lint`). In a way, the evolution of C recapitulates the history of programming languages, with its shift of emphasis from efficiency to portability and security.

Ritchie acknowledges that the language contains many infelicities, some of which result from attempting to maintain upward compatibility from B and BCPL; others are "historical accidents or mistakes." The first published description (*The C Programming Language* by Kernighan and Ritchie) was published in 1978, but it was not a language definition per se, since it was vague about many issues, and it was not consistent with the "reference compiler" (`pcc`, the portable C compiler). Many of these problems were solved by the development of an ANSI standard C, which began in 1983; it was approved in 1989.

During the late 1970s and early 1980s, use of Unix spread outside of AT&T, mostly within the university and industrial research communities, and C spread with it. By the late 1980s it had become a popular language for programming personal computers (for which, again, efficiency was often critical). It is also used as an output language for compilers of other languages (much as a structured assembler might be used), and has been the basis for other languages, such as "C++" (see Section 12.5).

C Mixes Characteristics of Three Generations

As a consequence of its history, C combines characteristics of several language generations. First, it has some third-generation features such as hierarchical data structures. Also it borrowed many ideas from second-generation languages, such as Algol-68, CPL, and PL/I. For example, its low-level model of arrays and pointers complicates or precludes optimization of array operations on some computers (in effect, C's storage model is lower level than the machine's). In some ways, C even returns to the first generation. For example, it does not permit nested procedures or environments. Thus it has poor support for modular programming (a key issue addressed by fourth-generation languages; see Chapter 7). It also resurrected some first-generation syntactic conventions, such as using '=' for assignment and requiring declarations to start with a keyword.

Perhaps we should not be surprised at the reappearance of first-generation characteristics in C; some, at least, were a direct consequence of its orientation to a machine, the 8K PDP-7, of comparable power to the machines for which the first-generation languages were designed. Thus we may say that C was motivated by concerns similar to those that motivated first-generation language designers.

Ritchie remarks that "C is quirky, flawed, and an enormous success." Aside from riding on the back of Unix popularity, he suggests that the success of C can be attributed to its simplicity, efficiency, portability, closeness to the machine, and its evolution in an environment in which it was used to write practical programs.

Characteristics of Third-Generation Programming Languages

The third-generation languages, as typified by Pascal, show an emphasis on simplicity and efficiency, and more generally a reaction to the excesses of the second generation. We consider individually the domains of data, name, and control structure.[7]

The *data structures* of the third generation show a shift of emphasis from the machine to the application. This is clearest in the provision of user-defined data types, which permit programmers to create the data types needed for their applications. It is also exemplified by application-oriented type constructors, such as sets, subranges, and enumeration types. The third generation is also characterized by the ability to nest data structures to any depth, that is, by the ability to organize data hierarchically.

The *name structures* of the third generation are generally some simplification of Algol-60 block structure. On the other hand, third-generation languages typically have new binding and scope-defining constructs, often associated with data type constructors, such as records and enumeration types.

Third-generation *control structures* are simplified, efficient versions of those found in the second generation. This is especially apparent in Pascal's **for**-loop, but it can also be seen in the rejection of name parameters and similar delayed-evaluation mechanisms. The third generation also has new control structures that are more application oriented, such as the **case**-statement.

In summary, the third generation combines practical engineering principles with the technical achievements of the second generation. The result, especially in the case of Pascal, is a simple, efficient, secure programming tool for many applications.

EXERCISES*

1**. Read about Euler (*Commun. ACM 9*, 1–2, 1966), Algol W (*Commun. ACM 9*, 6, 1966), and PL360 (*J. ACM 15*, 1, 1968). Trace the development of Wirth and Hoare's ideas from Algol-60, through Euler, Algol W, and PL360, to Pascal.

2**. Critique the language, Modula-2 (Wirth, *Programming in Modula-2*, 1982), which Wirth designed as a successor to Pascal.

3**. Study and evaluate Oberon, Wirth's successor to Modula-2 (Wirth, *Soft. Prac. Exper. 18*, 7, 1988, pp. 671–690).

4**. Study the *Proceedings* of the 1969 and 1971 symposia on extensible languages (*SIGPLAN Notices 4*, 8, 1969 and *SIGPLAN Notices 6*, 12, 1971). Write a report surveying the entire field at this time, or pick out one or more particular languages and write a detailed critique of them.

[7] The syntactic structures are essentially those of the second generation.

5.** Read, summarize, and critique Wirth's article, "On the Design of Programming Languages" (*Information Processing 74*, North-Holland, 1975).

6.** Pascal does not have a formatted input-output system. Decide whether formatted input-output should be part of Pascal or be provided by a procedure library, and design a corresponding input-output system.

7. A deficiency of Pascal's input-output system is that it does not allow input-output of values that are elements of enumeration types. Identify the problem with input-output of these values and design an input-output system that accommodates enumerations.

8. Pascal does not allow constants to be defined by an expression, even if the value of that expression is constant (see Section 5.4). Discuss this restriction thoroughly. What would be the benefits of allowing expressions in constant declarations? What restrictions, if any, should be placed on the allowable expressions? Are there other changes that could be made to the language to mitigate the absence of expressions in constant declarations?

9. All enumerations in Pascal are *ordered.* For example, with the enumeration declaration

```
type sex = (male, female);
```

we have male < female and **succ**(male) = female. These relationships really do not make any sense and the fact that Pascal permits them violates the Security Principle. One solution would be to add to Pascal another kind of enumeration type, an *unordered* enumeration. Discuss the trade-offs involved in deciding whether to add this feature to Pascal.

10. Suppose S is a set variable. If we wanted to perform some operation on each element of S, we would like to be able to write a **for**-loop like this:

```
for x in S do ....
```

Unfortunately, this is not legal Pascal. Instead we have to write a loop over all the elements of the base type of S and, for each element, test whether it is in S and perform the operation if it is. That is,

```
for x := least to greatest do
   if x in S then ....
```

What set operations could be added to Pascal to solve this problem?

11. Derive the addressing equation for Pascal arrays.

12. Propose a solution to the security problem posed by variant records.

13. Show how a Pascal program would have to be structured so that two arrays share the same memory locations.

14. To write a **case**-statement that does one thing for digits and another for letters, we would have to write something like this:

```
case ch of
    '0',   '1',   '2',   '3',   '4',
    '5',   '6',   '7',   '8',   '9':  ...do digit case...
    'a',   'b',   'c',   'd',   'e',
        'f',   'g',   'h',   'i',   'j',
        'k',   'l',   'm',   'n',   'o',
        'p',   'q',   'r',   's',   't',
        'u',   'v',   'w',   'x',   'y',
        'z':                                 ...do letter case...
end
```

Develop an extension to Pascal that avoids the error-prone and tedious process of listing all the digits and letters.

15. Some Pascal dialects permit an **otherwise** clause on **case**-statements. This clause is executed if the value of the case selector is not among the case labels. Write a BNF (or extended BNF) description of a **case**-statement with an **otherwise** clause.

16.** Read about the C language and evaluate it according to the principles discussed in this book.

17.** Read about CPL (D. W. Barron et al., "The Main Features of CPL." *Computer J. 6*, 2, 1963) and write a detailed evaluation of its features.

18.** Read and critique the criticisms and defenses of Pascal by Habermann (*Acta Inform. 3*, 1973, pp. 47–57), Lecarme and Desjardins (*Acta Inform. 4*, 1975, pp. 231–243), Welsh et al. (*Soft. Prac. Exper. 7*, 1977, pp. 685–696), and Kernighan ("Why Pascal Is Not My Favorite Programming Language." AT&T Bell Labs. Comp. Sci. Tech. Rep. 100, 1981).