Chapter 1

Introduction

The purpose of this book is to study the principles and innovations found in modern programming languages. We will consider a wide variety of languages. The goal is not to become proficient in any of these languages, but to learn what contributions each has made to the "state of the art" in language design.

I will discuss various programming paradigms in this book. Some languages (such as Ada, Pascal, Modula-2) are **imperative**; they use variables, assignments, and iteration. For imperative languages, I will dwell on such issues as flow of control (Chapter 2) and data types (Chapter 3). Other languages (for example, LISP and FP) are **functional**; they have no variables, assignments, or iteration, but model program execution as expression evaluation. I discuss functional languages in Chapter 4. Other languages (for example, Smalltalk and C++), represent the **object-oriented** paradigm, in which data types are generalized to collections of data and associated routines (Chapter 5). **Dataflow languages** (Val, Sisal, and Post, Chapter 6) attempt to gain speed by simultaneous execution of independent computations; they require special computer architectures. A more common way to gain speed is by **concurrent** programming (typified by languages such as SR and Lynx, discussed in Chapter 7). Another major paradigm constitutes the **declarative** languages such as Prolog and Gödel (Chapter 8); they view programming as stating what is wanted and not necessarily how to compute it. **Aggregate languages** (Chapter 9) form a a final loosely knit paradigm that includes languages with special-purpose data formats, such as strings (SNOBOL and Icon), arrays (APL), databases (dBASE and SQL), and mathematical formulas (Mathematica and Maple).

In addition to studying actual programming language constructs, I will present formal semantic models in Chapter 10. These models allow a precise specification of what a program means, and provide the basis for reasoning about the correctness of a program.

1 ◆ PROGRAMMING LANGUAGES AS SOFTWARE TOOLS

Programming languages fit into a larger subject that might be termed **software tools**. This subject includes such fields as interactive editors (text, picture, spreadsheet, bitmap, and so forth), data transformers (compilers, assemblers, stream editors, macro processors, text formatters), operating systems, database management systems, and tools for program creation, testing, and maintenance (script files, source-code management tools, debuggers).

In general, software tools can be studied as interfaces between clients, which are usually humans or their programs, and lower-level facilities, such as files or operating systems.

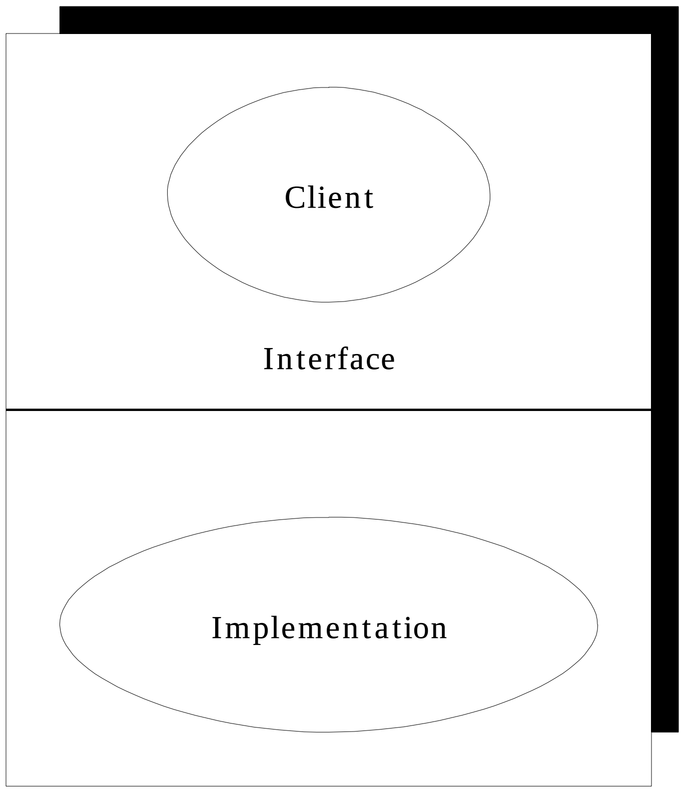


Figure 1.1 Software tools

Three questions arising from Figure 1.1 are worth discussing for any software tool:

1. What is the nature of the interface?
2. How can the interface be implemented by using the lower-level facilities?
3. How useful is the interface for humans or their agents?

When we deal with programming languages as software tools, these questions are transformed:

1. What is the structure (syntax) and meaning (semantics) of the programming language constructs? Usually, I will use informal methods to show what the constructs are and what they do. However, Chapter 10 presents formal methods for describing the semantics of programming languages.
2. How does the compiler writer deal with these constructs in order to translate them into assembler or machine language? The subject of compiler construction is large and fascinating, but is beyond the scope of this book. I will occasionally touch on this topic to assure you that the constructs can, in fact, be translated.
3. Is the programming language good for the programmer? More specifically, is it easy to use, expressive, readable? Does it protect the programmer from programming errors? Is it elegant? I spend a significant amount of effort trying to evaluate programming languages and their constructs in this way. This subject is both fascinating and difficult to be objective about. Many languages have their own fan clubs, and discussions often revolve about an ill-defined sense of elegance.

Programming languages have a profound effect on the ways programmers formulate solutions to problems. You will see that different paradigms impose very different programming styles, but even more important, they change the way the programmer looks at algorithms. I hope that this book will expand your horizons in much the same way that your first exposure to recursion opened up a new way of thinking. People have invented an amazing collection of elegant and expressive programming structures.

2 ◆ EVALUATING PROGRAMMING LANGUAGES

This book introduces you to some unusual languages and some unusual language features. As you read about them, you might wonder how to evaluate the quality of a feature or an entire language. Reasonable people disagree on what makes for a great language, which is why so many novel ideas abound in the arena of programming language design. At the risk of oversimplification, I would like to present a short list of desiderata for programming languages [Butcher 91]. Feel free to disagree with them. Another excellent discussion of this topic is found in Louden [Louden 93].

- **Simplicity.** There should be as few basic concepts as possible. Often the job of the language designer is to discard elements that are superfluous, error-prone, hard to read, or hard to compile. Many people consider PL/I, for example, to be much too large a language. Some criticize Ada for the same reason.
- **Uniformity.** The basic concepts should be applied consistently and universally. We should be able to use language features in different contexts without changing their form. Non-uniformity can be annoying. In Pascal, constants cannot be declared with values given by expressions, even though expressions are accepted in all other contexts when a value is needed. Non-uniformity can also be error-prone. In Pascal, some **for** loops take a single statement as a body, but **repeat** loops can take any number of statements. It is easy to forget to bracket multiple statements in the body of a **for** loop.

- **Orthogonality.** Independent functions should be controlled by independent mechanisms. (In mathematics, independent vectors are called "orthogonal.")
- **Abstraction.** There should be a way to factor out recurring patterns. (Abstraction generally means hiding details by constructing a "box" around them and permitting only limited inspection of its contents.)
- **Clarity.** Mechanisms should be well defined, and the outcome of code should be easily predictable. People should be able to read programs in the language and be able to understand them readily. Many people have criticized C, for example, for the common confusion between the assignment operator (=) and the equality test operator (==).
- **Information hiding.** Program units should have access only to the information they require. It is hard to write large programs without some control over the extent to which one part of the program can influence another part.
- **Modularity.** Interfaces between programming units should be stated explicitly.
- **Safety.** Semantic errors should be detectable, preferably at compile time. An attempt to add values of dissimilar types usually indicates that the programmer is confused. Languages like Awk and SNOBOL that silently convert data types in order to apply operators tend to be error-prone.
- **Expressiveness.** A wide variety of programs should be expressible.[1] Languages with coroutines, for example, can express algorithms for testing complex structures for equality much better than languages without coroutines. (Coroutines are discussed in Chapter 2.)
- **Efficiency.** Efficient code should be producible from the language, possibly with the assistance of the programmer. Functional programming languages that rely heavily on recursion face the danger of inefficiency, although there are compilation methods (such as eliminating tail recursion) that make such languages perfectly acceptable. However, languages that require interpretation instead of compilation (such as Tcl) tend to be slow, although in many applications, speed is of minor concern.

3 ◆ BACKGROUND MATERIAL ON PROGRAMMING LANGUAGES

Before showing you anything out of the ordinary, I want to make sure that you are acquainted with the fundamental concepts that are covered in an undergraduate course in programming languages. This section is intentionally concise. If you need more details, you might profitably refer to the fine books by Pratt [Pratt 96] and Louden [Louden 93].

[1] In a formal sense, all practical languages are Turing-complete; that is, they can express exactly the same algorithms. However, the ease with which a programmer can come up with an appropriate program is part of what I mean by expressiveness. Enumerating binary trees (see Chapter 2) is quite difficult in most languages, but quite easy in CLU.

3.1 Variables, Data Types, Literals, and Expressions

I will repeatedly refer to the following example, which is designed to have a little bit of everything in the way of types. A **type** is a set of values on which the same operations are defined.

Figure 1.2

```
variable                                                 1
     First : pointer to integer;                         2
     Second : array 0..9 of                              3
          record                                         4
               Third: character;                         5
               Fourth: integer;                          6
               Fifth : (Apple, Durian, Coconut, Sapodilla,  7
                    Mangosteen)                          8
          end;                                           9

begin                                                    10
     First := nil;                                       11
     First := &Second[1].Fourth;                         12
     Firstˆ := 4;                                        13
     Second[3].Fourth := (Firstˆ + Second[1].Fourth) *   14
          Second[Firstˆ].Fourth;                         15
     Second[0] := [Third : 'x'; Fourth : 0;              16
          Fifth : Sapodilla];                            17
end;                                                     18
```

Imperative languages (such as Pascal and Ada) have **variables**, which are named memory locations. Figure 1.2 introduces two variables, `First` (line 2) and `Second` (lines 3–9). Programming languages often restrict the values that may be placed in variables, both to ensure that compilers can generate accurate code for manipulating those values and to prevent common programming errors. The restrictions are generally in the form of type information. The **type** of a variable is a restriction on the values it can hold and what operations may be applied to those values. For example, the type `integer` encompasses numeric whole-number values between some language-dependent (or implementation-dependent) minimum and maximum value; values of this type may act as operands in arithmetic operations such as addition. The term `integer` is not set in bold monospace type, because in most languages, predefined types are not reserved words, but ordinary identifiers that can be given new meanings (although that is bad practice).

Researchers have developed various taxonomies to categorize types [ISO/IEC 94; Meek 94]. I will present here a fairly simple taxonomy. A **primitive type** is one that is not built out of other types. Standard primitive types provided by most languages include `integer`, `Boolean`, `character`, `real`, and sometimes `string`. Figure 1.2 uses both `integer` and `character`. Enumeration types are also primitive. The example uses an enumeration type in lines 7–8; its values are restricted to the values specified. Enumeration types often define the order of their enumeration constants. In Figure 1.2, however, it makes no sense to consider one fruit greater than another.[2]

Structured types are built out of other types. Arrays, records, and pointers are structured types.[3] Figure 1.2 shows all three kinds of standard structured types. The building blocks of a structured type are its **components**. The component types go into making the structured type; component values go into making the value of a structured type. The pointer type in line 2 of Figure 1.2 has one component type (integer); a pointer value has one component value. There are ten component values of the array type in lines 3–9, each of a record type. Arrays are usually required to be **homogeneous**; that is, all the component values must be of the same type. Arrays are indexed by elements of an **index type**, usually either a subrange of integers, characters, or an enumeration type. Therefore, an array has two component types (the base type and the index type); it has as many component values as there are members in the index type.

Flexible arrays do not have declared bounds; the bounds are set at runtime, based on which elements of the array have been assigned values. **Dynamic-sized** arrays have declared bounds, but the bounds depend on the runtime value of the bounds expressions. Languages that provide dynamic-sized arrays provide syntax for discovering the lower and upper bounds in each dimension.

Array **slices**, such as `Second[3..5]`, are also components for purposes of this discussion. Languages (like Ada) that allow array slices usually only allow slices in the last dimension. (APL does not have such a restriction.)

The components of the record type in lines 4–9 are of types `character` and `integer`. Records are like arrays in that they have multiple component values. However, the values are indexed not by members of an index type but rather by named **fields**. The component values need not be of the same type; records are not required to be homogeneous. Languages for systems programming sometimes allow the programmer to control exactly how many bits are allocated to each field and how fields are packed into memory.

The **choice** is a less common structured type. It is like a record in that it has component types, each selected by a field. However, it has only one component value, which corresponds to exactly one of the component types. Choices are often implemented by allocating as much space as the largest component type needs. Some languages (like Simula) let the programmer restrict a variable to a particular component when the variable is declared. In this case, only enough space is allocated for that component, and the compiler disallows accesses to other components.

Which field is active in a choice value determines the operations that may be applied to that value. There is usually some way for a program to determine at runtime which field is active in any value of the choice type; if not, there is a danger that a value will be accidentally (or intentionally) treated as belonging to a different field, which may have a different type. Often, languages provide a **tagcase** statement with branches in which the particular variant is known both to the program and to the compiler. Pascal allows part

[2] In Southeast Asia, the durian is considered the king of fruits. My personal favorite is the mangosteen.

[3] Whether to call pointers primitive or structured is debatable. I choose to call them structured because they are built from another type.

of a record to be a choice and the other fields to be active in any variant. One of the latter fields indicates which variant is in use. It doesn't make sense to modify the value of that field without modifying the variant part as well.

A **literal** is a value, usually of a primitive type, expressly denoted in a program. For example, 243 is an integer literal and Figure 1.2 has literals `0`, `1`, `3`, `4`, `9`, and `'x'`. Some values are provided as predeclared constants (that is, identifiers with predefined and unchangeable values), such as `false` (Boolean) and `nil` (pointer).

A **constructor** expressly denotes a value of a structured type. Figure 1.2 has a record constructor in lines 16–17.

An **expression** is a literal, a constructor, a constant, a variable, an invocation of a value-returning procedure, a conditional expression, or an operator with operands that are themselves expressions. Figure 1.2 has expressions in lines 11–17. An **operator** is a shorthand for an invocation of a value-returning procedure whose parameters are the operands. Each operator has an **arity**, that is, the number of operands it expects. Common arities are unary (one operand) and binary (two operands). Unary operators are commonly written before their operand (such as -4 or `&myVariable`), but some are traditionally written after the operand (such as `ptrVariable^`). Sometimes it is helpful to consider literals and constants to be nullary (no-operand) operators. For example, `true` is a nullary operator of type **Boolean**.

Operators do not necessarily take only numeric operands. The **dereferencing** operator (`^`), for example, produces the value pointed to by a pointer. This operator is unary and **postfix**, that is, it follows its expression. You can see it in Figure 1.2 in lines 13, 14, and 15. Some languages, such as Gedanken, Ada, and Oberon-2, coerce pointers (repeatedly, if needed) to the values they dereference if the context makes it clear which type is required. The unary prefix **referencing** operator (&) in line 12 generates a pointer to a value.

Common operators include those in the table on the next page. Many operators are **overloaded**; that is, their meaning depends on the number and types of the operands. It is easiest to understand overloaded operators as multiply defined procedures, from which the compiler chooses the one with the appropriate number and type of parameters.

Each operator has an assigned **precedence**, which determines the way the expression is grouped in the absence of parentheses. In Figure 1.2, lines 14–15, the meaning would probably be different without the parentheses, because multiplication is usually given a higher precedence than addition.

Operator	Left type	Right type	Result type	Comments
`+ - *`	`integer`	`integer`	`integer`	
`+ - * /`	`real`	`real`	`real`	
`/`	`integer`	`integer`	`real`	or `integer`
div mod	`integer`	`integer`	`integer`	
`-`	`numeric`	`none`	`same`	
`**`	`integer`	`integer`	`integer`	exponentiation
`**`	`numeric`	`real`	`real`	exponentiation
`=`	`any`	`same`	`Boolean`	
`< > >= <=`	`numeric`	`same`	`Boolean`	
`+`	`string`	`string`	`string`	concatenation
`~`	`string`	`pattern`	`Boolean`	string match
and	`Boolean`	`Boolean`	`Boolean`	
or	`Boolean`	`Boolean`	`Boolean`	
not	`Boolean`	`none`	`Boolean`	
`^`	`pointer`	`none`	`component`	
`&`	`any`	`none`	`pointer`	

Expressions evaluate to **R-values**. Variables and components of variables of structured types also have an **L-value**, that is, an address where their R-value is stored. The assignment statement (lines 11–17 in Figure 1.2) requires an L-value on the left-hand side (*L* stands for "left") and an R-value on the right-hand side (*R* stands for "right"). In Figure 1.2, lines 11 and 12 show a variable used for its L-value; the next lines show components used for their L-values.

The types of the left-hand side and the right-hand side must be **assignment-compatible**. If they are the same type, they are compatible. (What it means to have the same type is discussed in Chapter 3.) If they are of different types, the language may allow the value of the right-hand side to be implicitly converted to the type of the left-hand side. Implicit type conversions are called **coercions**. For example, Pascal will coerce integers to reals, but not the reverse. Coercions are error-prone, because the target type may not be able to represent all the values of the source type. For example, many computers can store some large numbers precisely as integers but only imprecisely as reals.

Converting types, either explicitly (**casting**) or implicitly (**coercing**) can sometimes change the data format. However, it is sometimes necessary to treat an expression of one type as if it were of another type without any data-format conversion. For example, a message might look like an array of characters to one procedure, whereas another procedure must understand it as a record with header and data fields. Wisconsin Modula introduced a nonconverting casting operator **qua** for this purpose. In C, which lacks such an operator, the programmer who wishes a nonconverting cast must cast a pointer to the first type into a pointer to the second type; pointers have the same representation no matter what they point to (in most C implementations). The following code shows both methods.

Figure 1.3

```
type                                                      1
     FirstType = ... ;                                    2
     SecondType = ... ;                                   3
     SecondTypePtr = pointer to SecondType;               4
variable                                                  5
     F : FirstType;                                       6
     S : SecondType;                                      7
begin                                                     8
     ...                                                  9
     S := F qua SecondType; -- Wisconsin Modula          10
     S := (SecondTypePtr(&F))^; -- C                     11
end;                                                     12
```

Line 10 shows how F can be cast without conversion into the second type in Wisconsin Modula. Line 11 shows the same thing for C, where I use the type name SecondTypePtr as an explicit conversion routine. The referencing operator & produces a pointer to F. In both cases, if the two types disagree on length of representation, chaos may ensue, because the number of bytes copied by the assignment is the appropriate number for SecondType.

The Boolean operators **and** and **or** may have **short-circuit** semantics; that is, the second operand is only evaluated if the first operand evaluates to `true` (for **and**) or `false` (for **or**). This evaluation strategy is an example of **lazy evaluation**, discussed in Chapter 4. Short-circuit operators allow the programmer to combine tests, the second of which only makes sense if the first succeeds. For example, I may want to first test if a pointer is `nil`, and only if it is not, to test the value it points to.

Conditional expressions are built with an **if** construct. To make sure that a conditional expression always has a value, each **if** must be matched by both a **then** and an **else**. The expressions in the **then** and **else** parts must have the same type. Here is an example:

Figure 1.4

```
write(if a > 0 then a else -a);
```

3.2 Control Constructs

Execution of imperative programming languages proceeds one **statement** at a time. Statements can be **simple** or **compound**. Simple statements include the assignment statement, procedure invocation, and **goto**. Compound statements enclose other statements; they include conditional and iterative statements, such as **if**, **case**, **while**, and **for**. Programming languages need some syntax for delimiting enclosed statements in a compound statement. Some languages, like Modula, provide closing syntax for each compound statement:

Figure 1.5

```
while First^ < 10 do                                      1
     First^ := 2 * First^;                                2
     Second[0].Fourth := 1 + Second[0].Fourth;            3
end;                                                      4
```

The **end** on line 4 closes the **while** on line 1. Other languages, like Pascal, only allow a single statement to be included, but it may be a **block state-**

ment that encloses multiple statements surrounded by **begin** and **end**.

Syntax for the **if** statement can be confusing if there is no trailing **end** syntax. If an **if** statement encloses another **if** statement in its **then** clause, the **else** that follows might refer to either **if**. This problem is called the "dangling-**else**" problem. In the following example, the **else** in line 4 could match either the one in line 1 or line 2. Pascal specifies that the closer **if** (line 2) is used.

Figure 1.6

```
if IntVar < 10 then                  1
if IntVar < 20 then                  2
     IntVar := 0                     3
else                                 4
     IntVar := 1;                    5
```

On the other hand, if **if** statements require a closing **end**, the problem cannot arise:

Figure 1.7

```
if IntVar < 10 then                  1
     if IntVar < 20 then             2
          IntVar := 0;               3
     end                             4
else                                 5
     IntVar := 1;                    6
end;                                 7
```

Here, the **else** in line 5 unambiguously matches the **if** in line 1. Closing syntax is ugly when **if** statements are deeply nested in the **else** clause:

Figure 1.8

```
if IntVar < 10 then                  1
     IntVar := 0                     2
else                                 3
     if IntVar < 20 then             4
          IntVar := 1                5
     else                            6
          if IntVar < 30 then        7
               IntVar := 2           8
          else                       9
               IntVar := 3;          10
          end;                       11
     end;                            12
end;                                 13
```

The **elsif** clause clarifies matters:

Figure 1.9

```
if IntVar < 10 then
     IntVar := 0
elsif IntVar < 20 then
     IntVar := 1
elsif IntVar < 30 then
     IntVar := 2
else
     IntVar := 3;
end;
```

All the examples in this book use a closing **end** for compound statements. You don't have to worry about language-specific syntax issues when you are trying to concentrate on semantics.

Some languages, like Russell and CSP, allow conditionals to have any number of branches, each with its own Boolean condition, called a **guard**. The guards may be evaluated in any order, and execution chooses any branch whose guard evaluates to `true`. These conditionals are called **nondeterministic**, since running the program a second time with the same input may result in a different branch being selected. In such languages, **else** means "when all the guards are `false`."

A wide range of iterative statements (loops) is available. An iterative statement must indicate under what condition the iteration is to terminate and when that condition is tested. The **while** loop tests an arbitrary Boolean expression before each iteration.

When **goto** statements became unpopular because they lead to unreadable and unmaintainable programs, languages tried to avoid all control jumps. But loops often need to exit from the middle or to abandon the current iteration and start the next one. The **break** and **next** statements were invented to provide these facilities without reintroducing unconstrained control jumps. An example of exiting the loop from the middle is the "*n*-and-a-half-times loop":

Figure 1.10

```
loop                                            1
     read(input);                               2
     if input = 0 then break end;               3
     if comment(input) then next end;           4
     process(input);                            5
end;                                            6
```

The **break** in line 3 terminates the loop when a sentinel indicating the end of input is read. The **next** in line 4 abandons the current iteration if the input is not to be processed. A similar statement found in Perl is **redo**, which restarts the current iteration without updating any loop indices or checking termination conditions. The **break**, **next**, and **redo** statements can also take an integer or a loop label to specify the number of levels of loop they are to terminate or iterate. In this case, they are called **multilevel** statements.

Many loops require control variables to be initialized before the first iteration and updated after each iteration. Some languages (like C) provide syntax that includes these steps explicitly, which makes the loops more readable and less error-prone. However, such syntax blurs the distinction between definite (**for**) and indefinite (**while**) iteration:

Figure 1.11

```
for a := 1; Ptr := Start -- initialization
while Ptr ≠ nil -- termination condition
updating a := a+1; Ptr := Ptr^.Next; -- after each iter.
do
    ... -- loop body
end;
```

Russell and CSP generalize the nondeterministic **if** statement into a nondeterministic **while** loop with multiple branches. So long as any guard is true, the loop is executed, and any branch whose guard is true is arbitrarily selected and executed. The loop terminates when all guards are false. For example, the algorithm to compute the greatest common divisor of two integers a and b can be written as follows:

Figure 1.12

```
while
    when a < b => b := b - a;
    when b < a => a := a - b;
end;
```

Each guard starts with the reserved word **when** and ends with the symbol => . The loop terminates when a = b.

The **case** statement is used to select one of a set of options on the basis of the value of some expression.[4] Most languages require that the selection be based on a criterion known at compile time (that is, the case labels must be constant or constant ranges); this restriction allows compilers to generate efficient code. However, conditions that can only be evaluated at runtime also make sense, as in the following example:

Figure 1.13

```
case a of                                                    1
    when 0 => Something(1); -- static unique guard          2
    when 1..10 => Something(2); -- static guard             3
    when b+12 => Something(3); -- dynamic unique guard      4
    when b+13..b+20 => Something(4); -- dynamic guard       5
    otherwise Something(5); -- guard of last resort         6
end;                                                         7
```

Each guard tests the value of a. Lines 2 and 4 test this value for equality with 0 and b+12; lines 3 and 5 test it for membership in a range. If the guards (the selectors for the branches) overlap, the **case** statement is erroneous; this situation can be detected at compile time for static guards and at runtime for dynamic guards. Most languages consider it to be a runtime error if none of the branches is selected and there is no **otherwise** clause.

[4] C. A. R. Hoare, who invented the **case** statement, says, "This was my first programming language invention, of which I am still most proud." [Hoare 73]

3.3 Procedures and Parameter Passing

Figure 1.14 will be discussed in detail in this section. For clarity, I have chosen a syntax that names each formal parameter at the point of invocation; Ada and Modula-3 have a similar syntax.

Figure 1.14

```
procedure TryAll(                                                  1
     ValueInt : value integer;                                     2
     ReferenceInt : reference integer;                             3
     ResultInt : result integer;                                   4
     ReadOnlyInt : readonly integer := 10;                         5
     NameInt : name integer;                                       6
     MacroInt : macro integer) : integer;                          7
variable                                                           8
     LocalInt : integer;                                           9
begin                                                              10
     LocalInt := 10; -- affects only TryAll's LocalInt             11
     ValueInt := 1 + ValueInt; -- formal becomes 16                12
     ReferenceInt := 1 + ValueInt;                                 13
          -- actual and formal become 17                           14
     ResultInt := 1 + ReferenceInt + ReadOnlyInt + NameInt;        15
          -- 47                                                    16
     return 2*MacroInt; -- 40                                      17
end; -- TryAll                                                     18

variable                                                           19
     LocalInt : integer;                                           20
     A, B : integer;                                               21

begin -- main program                                              22
     LocalInt := 3;                                                23
     B := TryAll(                                                  24
          ValueInt : 15,                                           25
          ReferenceInt : LocalInt,                                 26
          ResultInt : A, -- becomes 47                             27
          ReadOnlyInt  : 12,                                       28
          NameInt : LocalInt,                                      29
          MacroInt : 2*LocalInt)                                   30
     );                                                            31
     -- Final values: LocalInt = 17, A = 47, B = 40                32
end; -- main program                                               33
```

Procedures (often called **functions** if they return values) are usually declared with a header, local declarations, and a body. The **header** (lines 1–7) indicates the procedure name and the parameters, if any, along with their types and modes. If the procedure is to return a value, the type of the value is also declared. If not, the predeclared type `void` is used in some languages to indicate that no value at all is returned. The declarations (lines 8 and 9) introduce local meanings for identifiers. Together, the parameters and the local identifiers constitute the **local referencing environment** of the procedure. Identifiers appearing within the procedure are interpreted, if possible, with respect to the local referencing environment. Otherwise, they are interpreted with respect to parts of the program outside the procedure. The nonlocal referencing environment is more complicated, so I will discuss it later.

The **body** of the procedure (lines 10–18) is composed of the statements that are to be executed when the procedure is invoked. The header need not be adjacent to the declarations and body; they may be separated for the purpose of modularization (discussed in Chapter 3). Most programming languages allow **recursion**; that is, procedures may invoke themselves, either directly or indirectly.

Parameters are inputs and outputs to procedures. The identifiers associated with parameters in the header are called **formal parameters**; the expressions passed into those parameters at the point of invocation (lines 24–31) are the **actual parameters**. There are many parameter-passing **modes**, each with different semantics specifying how formal parameters are bound to actual parameters.

- **Value.** The value of the actual parameter is copied into the formal parameter at invocation. In the example, the assignment in line 12 modifies the formal, but not the actual parameter; the expression in line 13 uses the modified value in the formal. Value mode is the most common parameter-passing mode. Some languages, like C, provide only this mode.
- **Result.** The value of the formal parameter is copied into the actual parameter (which must have an L-value) at procedure return. In the example, the assignment in line 15 gives the formal a value, which is copied into the actual parameter `A` (line 27) when the procedure `TryAll` returns. It is usually invalid to provide actual parameters with the same L-value to two different result parameters, because the order of copying is undefined. However, this error cannot always be caught by the compiler, because it cannot always tell with certainty that two identifiers will have different L-values at runtime.
- **Value result.** The parameter is treated as in value mode during invocation and as in result mode during return.
- **Reference.** The L-value of the formal parameter is set to the L-value of the actual parameter. In other words, the address of the formal parameter is the same as the address of the actual parameter. Any assignment to the formal parameter immediately affects the actual parameter. In the example, the assignment in line 13 modifies both the formal parameter (`ReferenceInt`) and the actual parameter (`LocalInt` of the main program), because they have the same L-value. Reference mode can be emulated by value mode if the language has a referencing operator (I use &), which produces a pointer to an expression with an L-value, and a dereferencing operator (I use ˆ), which takes a pointer and produces the value pointed to. The program passes the pointer in value mode and dereferences the formal parameter every time it is used. FORTRAN only has reference mode; expressions, which have no L-value, are evaluated and placed in temporary locations in order to acquire an L-value for the duration of the procedure.[5] Large arrays are usually passed in reference mode instead of value mode to avoid the copying otherwise required.

[5] Some implementations of FORTRAN store all literals in a data region at runtime. A literal actual parameter is at risk of being modified by the procedure, after which the literal will have a new value!

- **Readonly.** Either value or reference mode is actually used, but the compiler ensures that the formal parameter is never used on the left-hand side of an assignment. The compiler typically uses value mode for small values (such as primitive types) and reference mode for larger values. In the example, it would be invalid for `ReadOnlyInt` to be used on the left-hand side of the assignment on line 15.

The following modes have been proposed and used in the past, but are no longer in favor due to their confusing semantics and difficult implementation.

- **Name.** Every use of the formal parameter causes the actual parameter to be freshly evaluated in the referencing environment of the invocation point. If the formal parameter's L-value is needed (for example, the parameter appears on the left-hand side of an assignment), the actual parameter's L-value must be freshly evaluated. If the formal parameter's R-value is needed, the actual parameter's R-value must be freshly evaluated. This mode is more complex than reference mode, because the actual parameter may be an expression, and the procedure may modify one of the variables that make up that expression. Such a modification affects the value of the formal parameter. In the example, `NameInt` in line 15 evaluates to `LocalInt` of the main program, which was modified by the assignment in line 13. Name mode was invented in Algol 60, caused a certain amount of consternation among compiler writers, who had to invent an implementation, and proved to be not very useful and fairly error-prone.[6] Modern languages don't usually provide name mode.
- **Macro.** Every use of the formal parameter causes the text of the actual parameter to be freshly evaluated in the referencing environment of the use point. That is, if the actual parameter is a variable, `IntVar`, and the procedure declares a new variable with the same name, then reference to the formal parameter is like reference to the new, not the old, `IntVar`. In the example, `MacroInt` in line 17 expands to `2*LocalInt`, the actual parameter (line 30), but `LocalInt` is interpreted as referring to the variable belonging to `TryAll`, not to the main program. Macro mode is extremely error-prone, not very useful, and almost never provided. It opens the possibility of runtime parsing, because the actual parameter could be an expression fragment, such as `+ LocalInt`, which would need to be understood in the syntactic context of each use of the formal parameter.

Procedures themselves may be passed as parameters. In this case, we generally don't talk about the parameter-passing mode.[7] The formal parameter declaration may indicate the number and types of the parameters to the passed procedure. The formal parameter may be used in any way that a procedure can be used: it can be invoked or passed again as an actual parameter.

`Goto` labels may also be passed as parameters. The formal parameter may then be the target of a **`goto`** or may be passed again as an actual parameter. If it is the target of a **`goto`**, the referencing environment of the original in-

[6] J. Jensen invented a clever use for name-mode parameters that is called "Jensen's device", but its cleverness is outweighed by its lack of clarity.

[7] You might say that the procedure is passed by value, but in fact, no copy is made. Instead, a closure is passed; this concept is elaborated below and in Chapter 3.

voker is restored, and intervening referencing environments are closed. (I will discuss referencing environments shortly.) Implementing these semantics correctly is complex, and few languages with block structure allow labels to be passed as parameters.

Sometimes the programmer cannot predict how many parameters will be provided. This situation arises particularly for input and output routines. If there may be an arbitrary number of actual parameters of the same type, they may be packaged into an array (perhaps an anonymous dynamic-sized array built by a constructor). The formal parameter can be queried to discover how many elements were passed.

Ada, C++, and Common LISP provide default values, so that formal parameters that have no matching actuals can still have values; line 5 in Figure 1.14 provides a default value of 10 for parameter `ReadOnlyInt` in case it is not provided by the call. A call can just omit an actual parameter to indicate that it is missing. Only trailing parameters (that is, the parameters at the end of the parameter list) may be omitted, so that the compiler can determine which ones are missing. Other syntax is possible. For example, the procedure call could still delimit missing parameters with commas (such as `myProcedure(paramA,,paramC)`). Alternatively, the call may explicitly associate formal and actual parameters in any order. Lines 24–31 in Figure 1.14 use this **keyword** (as opposed to **positional**) parameter-passing syntax for specifying actuals. Keyword parameters make it easy to omit an actual parameter.

Languages differ in the syntax they use to return a value to the caller. Line 17 of Figure 1.14 shows **explicit return**, in which the **return** statement includes the value. The compiler can check that all returns specify a value of the appropriate type and that the procedure does not terminate without returning a value. Often the programmer introduces a local variable to construct and manipulate the value before returning it; the actual return results in an extra copy step. **Implicit return** uses the procedure identifier as a write-only pseudovariable that is automatically returned when the procedure finishes. The compiler cannot check that all execution paths set this variable, and the programmer must be careful not to use the procedure identifier as an ordinary variable, because such use may be misunderstood as a recursive procedure invocation. If the procedure needs to manipulate the value before it is finalized, programmers usually introduce a local variable and copy it into the write-only variable. Finally, **identifier return** introduces a new identifier or identifiers in the procedure header to represent the returned values, as in the following example:

Figure 1.15

```
procedure Double(                                          1
     ValueInt : value integer) : integer RetVal;           2
begin                                                      3
     RetVal := ValueInt * 2;                               4
     if RetVal < 0 then RetVal := 0; end;                  5
end; -- Double                                             6
```

Line 2 introduces the new identifier `RetVal`, and line 4 assigns it a value. Line 5 treats it as an ordinary integer variable. Neither the program nor the compiled code needs to copy the values from the new identifiers into return-value cells.

The new-identifier method makes it easy to describe procedures that return multiple values. Such procedures are invoked in a context of multiple assignment, as in Figure 1.16. Here, procedure TwoVals returns two results, which are assigned simultaneously to two variables in the multiple assignment of line 8.

Figure 1.16

```
procedure TwoVals : integer Answer1, Answer2;                  1
begin                                                          2
    Answer1 := 3;                                              3
    Answer2 := 9;                                              4
end;                                                           5

variable a, b : integer;                                       6

begin                                                          7
    a, b := TwoVals;                                           8
end;                                                           9
```

3.4 Block Structure

I will describe classic Algol block structure here; it has been adopted, with modification, in many programming languages. A program is divided into nested **blocks**, each of which introduces a new name scope. A **name scope** is a region of program in which particular declarations of identifiers are in effect. A declaration **maps** an identifier to a meaning. We also say that it **binds** the meaning to the identifier. The meanings can be variables, types, constants, labels, procedures, or other concepts discussed elsewhere in the book, such as modules (Chapter 3), classes (Chapter 5), and monitors (Chapter 7). Traditionally, each nested name scope inherits all bindings from the surrounding scope, except that if the same identifier is redefined in the nested scope, the new declaration **overrides** the old declaration for the duration of the nested scope. Some languages, such as Ada and C++, allow declared procedures to be overloaded; that is, the same name is bound to multiple declarations at the same time, and the compiler chooses which is meant by the number and types of the parameters.

The new declarations can be defined to take effect from the beginning of the block (so that an earlier declaration, say of a variable, can refer to a later declaration, perhaps of a type). More commonly, they take effect (are **elaborated**) from the point in the block where the declaration appears. In the following example, I could define B in line 8 to be either real or integer, depending on whether the outer declaration of T is hidden yet by the declaration in line 10. Usually, languages either disallow such references or let the new declaration take effect only after the point at which it appears. This decision makes one-pass compilers easier to write.

Figure 1.17

```
type -- introduces outer block                                 1
    T : real;                                                  2
variable -- continues outer block                              3
    A : integer;                                               4
```

```
begin -- statements start                                    5
     A := 4;                                                 6
     variable -- introduces nested block                     7
          B : T; -- real or integer?                         8
     type                                                    9
          T : integer; -- overrides outer declaration of T   10
     begin                                                   11
          B := 3; -- coercion needed?                        12
     end -- nested block ends                                13
end -- block ends                                            14
```

I use **type**, **variable**, or **constant** to introduce a new block, which includes a new name scope (lines 1 and 7). After declarations introduce new identifiers (including multiple instances of **type**, **variable**, or **constant**), the statements in the name scope are delimited by **begin** and **end**.

Variables may be initialized to the value of some expression at the same time they are declared. Pascal restricts initialization expressions to literals and constants. Some languages allow arbitrary initialization expressions to be evaluated at elaboration time; these expressions may even invoke procedures.

Entering a new block just to introduce temporary declarations can be helpful in structuring programs. More commonly, though, blocks are found as the bodies of procedures. The identifiers introduced in the new block are all the formal parameters and any types, constants, variables, labels, and procedures defined within the procedure. A language is considered **block-structured** if procedures introducing name scopes can nest. By this criterion, C is not block-structured, but Pascal is.

An identifier is considered **local** to a name scope if it is introduced in that name scope. Identifiers inherited from surrounding scopes are called **nonlocal**. An identifier is **global** if it belongs to the outermost block of the program. In FORTRAN, there are no global identifiers, and name scopes do not nest. These restrictions help make FORTRAN efficient at runtime.

Although the *declaration* of an identifier may be clear from its defining name scope, the *instance* of the identifier may not be. Every invocation of a procedure introduces not only a new name scope, but also new instances of variables themselves.[8] A procedure may have many simultaneous instances, because it may be invoked recursively. For local identifiers and global identifiers, it is always clear which instance to use. For nonlocal identifiers, the **nonlocal referencing environment** refers to the set of identifier bindings dynamically in force during program execution. This set changes at every procedure invocation and return, as well as when the program enters and exits its blocks, as illustrated in the following example.

[8] Although this discussion centers on variables, it also applies to labels and types, because types may depend on runtime values. For example, an array type may have limits that are taken from runtime values.

Figure 1.18

```
procedure ProcA(value AParam : integer);
type AType : array 1..AParam of integer;
variable AVar1, AVar2 : integer;
     procedure ProcB(value BParam : integer);
     variable BVar1 : AType;
     begin -- ProcB
          ... -- some statements
     end; -- ProcB
begin -- ProcA
     ... -- some statements
end; -- ProcA
```

When ProcA is invoked, the new instance of ProcA elaborates a new set of formal parameters (AParam), types (AType), variables (AVar1 and AVar2), and procedures (ProcB), which are inherited by nested procedure ProcB. When ProcB is invoked, its new instance elaborates a new formal parameter (BParam) and variable (BVar1), the latter of a type inherited from ProcA. ProcB may be invoked many times by ProcA and ProcB; each time, its new instance inherits identifiers from the ProcA instance that elaborates the particular ProcB that is invoked.

The situation becomes surprisingly complex when procedures (and labels) are passed as parameters. They carry with them their nonlocal referencing environment, so that when they are invoked, they may access nonlocal variables that are otherwise inaccessible in the program. A procedure in combination with its nonlocal referencing environment is called a **closure**.

Because this idea is unfamiliar to students who mainly use C (which has no nested procedures, and therefore no nonlocal referencing environments), I will present several examples.

Figure 1.19

```
procedure A(procedure X());                          1
variable Z : integer;                                2
begin -- A                                           3
     X();                                            4
end; -- A                                            5

procedure B(S : integer);                            6
variable Z : integer;                                7
     procedure C();                                  8
     begin -- C                                      9
          write(Z); -- from lexical parent B        10
     end; -- C                                      11
begin -- B                                          12
     Z := S;                                        13
     C();                                           14
     A(C);                                          15
end; -- B                                           16

B(3);                                               17
```

When B is called in line 17, it sets its local variable Z (line 7) to 3, the value of formal parameter S. It then calls nested procedure C two times. The first time is a direct call (line 14), and the second is indirect through a helper procedure A, which just relays the call (lines 15 and 4). In both cases, B is still present, in the sense that its activation record is still on the central stack. Procedure C needs B's activation record, because C refers to B's local variable Z, which is only to be found in B's activation record. In fact, C must access B's copy of Z during the second call, even though the intermediate procedure A also has declared a local variable Z. In other words, C's nonlocal referencing environment is B, which elaborated C. When C is passed as an actual parameter to A in line 15, a closure must be passed, so that when A invokes its formal parameter X (which is actually C), the procedure it invokes can properly access its nonlocal variables.

Figure 1.20 shows the stack of invocations at the point C is invoked via A. The first row shows that the main program has declarations for A and B. The second row shows that B has been invoked, and that it has local identifiers S (the formal parameter, with actual value 3), Z (a locally declared integer), and C (a locally declared procedure). The third row shows that A has been invoked (from line 15 of the program). It has a formal parameter X (bound to the actual parameter C) and a local integer variable Z. The last row shows that A has called its formal parameter, which we know is procedure C from row 2. The arrows to the left of the box indicate the nonlocal referencing environment of each invocation. Rows 2 and 3 (B and A) both use the main program as their nonlocal referencing environment. Row 4, however, shows that C uses B as its nonlocal referencing environment. This is because C was elaborated first in B, as the connecting lines indicate. That is why when C finally refers to Z in line 10, it accesses the Z of the second row, the one belonging to B.

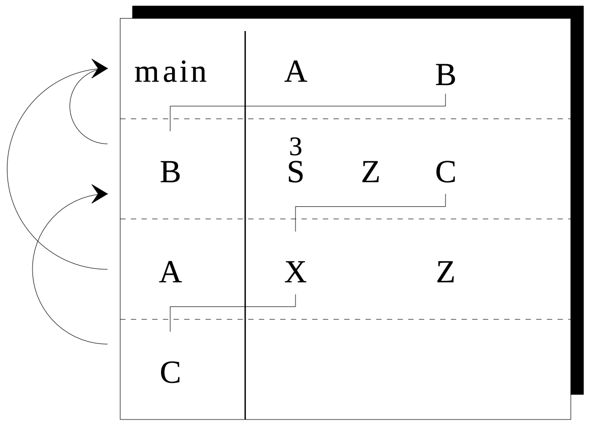


Figure 1.20 Referencing environments

The following example shows a more complicated situation.

Figure 1.21

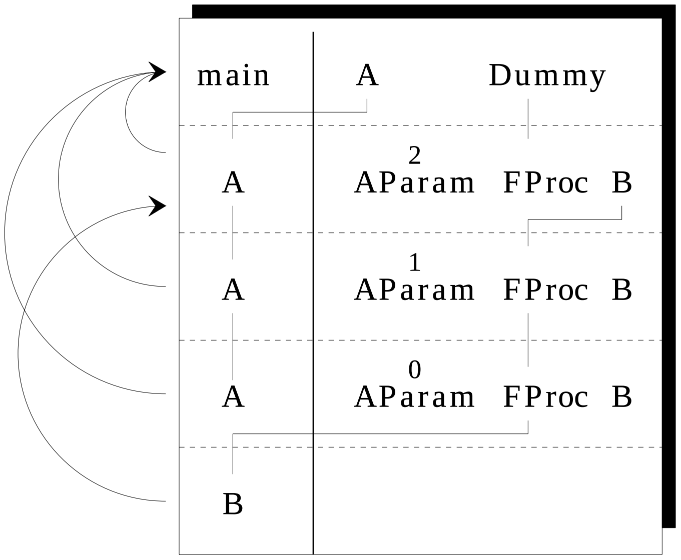
```
procedure A(
     readonly AParam : integer;
     AProc : procedure()
);
     procedure B();
     begin -- B
          write(AParam); -- writes 2
     end; -- B
begin -- A
     case AParam of
          when 2 => A(1, B);
          when 1 => A(0, AProc);
          when 0 => AProc();
     end; -- case
end; -- A

procedure Dummy(); begin end;
     -- never called; same type as B

begin -- main program
     A(2, Dummy);
end;
```

The referencing environments of each instance of each procedure are shown in Figure 1.22.

Figure 1.22 Referencing environments



Each row again shows an invocation of some procedure, starting with main. The entries on the row indicate the local referencing environment elaborated by that invocation. The arrows on the left indicate the nonlocal referencing environments. Here, main introduces A and Dummy. The instance of A that main invokes is the one it elaborated, as shown by the connecting line. Procedure A elaborates its parameters, AParam and AProc, and its nested procedure, B. When A invokes itself recursively, it uses the meaning of A in its nonlocal referencing environment, that is, the first row. It passes the closure of its own elaborated `B` as an actual parameter. This closure of `B` includes the nonlocal referencing environment of the first `A`, so when it is finally invoked, after being passed once more as a parameter to the third instance of `A`, it still owns the first `A` as its nonlocal referencing environment. When `B` prints `AParam`, therefore, it prints 2.

Binding the nonlocal referencing environment of a procedure at the time it is elaborated is called **deep binding**. Under deep binding, when a procedure is passed as a parameter, its closure is actually passed. The opposite, **shallow binding**, is to bind the nonlocal referencing environment of a procedure at the time it is invoked. Shallow binding does not pass closures. Descendents of Algol use deep binding; original LISP used shallow binding, although it provided a way for the programmer to explicitly build a closure.

Another difference between the Algol family and original LISP is the **scope rules** they follow to determine which syntactic entity is bound to each identifier. Languages in the Algol family are statically scoped, whereas original LISP was dynamically scoped.[9] Under **static scope rules**, the compiler can determine the declaration (although not necessarily the instance, as you have seen) associated with each identifier. The strict compile-time nesting of name scopes in Algol makes it a statically scoped language. In contrast, **dynamic scope rules** make identifiers accessible in a procedure if they were accessible at the point of invocation; therefore, different invocations can lead to different sets of accessible identifiers. The compiler cannot tell which identifiers are accessible to any procedure. The trend in programming language design has been away from dynamic scope rules, because they are often confusing to the programmer, who cannot tell at a glance which declaration is associated with each use of a variable. However, some recent languages, such as Tcl, use dynamic scope rules.

3.5 Runtime Store Organization

Programmers usually don't care how runtime store is organized. They expect the compiler or interpreter to arrange the program and data for efficient execution. They are only interested if some language constructs are likely to use large amounts of space or time. However, language designers are definitely interested in runtime store organization because it affects the efficient implementation of the language.

Runtime store is typically divided into several regions. The first region holds the compiled program instructions, which I will just call **code**. This region contains each procedure in the program as well as runtime libraries. Under some operating systems, the libraries may be shared among processes and may be brought into store dynamically when they are first referenced.

A second region holds global variables. Because the compiler knows the identity, type, and size of these variables, it can allocate precise amounts of store and can generate code that accesses global variables very efficiently.

A third region is the **central stack**. It holds an activation record for each active procedure instance. Because procedures are invoked and return in

[9] More recent LISP languages, such as Common LISP and Scheme, are statically scoped.

last-in, first-out order, a stack is appropriate. Each **activation record** stores the return address, a pointer to the activation record of its invoker (forming the **dynamic chain**), a pointer to the activation record of its nonlocal referencing environment (forming the **static chain**), its parameters, its local variables, and temporary locations needed during expression evaluation. It is possible to represent the static chain in various ways; for simplicity, I will just assume that it is a linked list of activation records. Dynamic-sized arrays are typically represented by a fixed-size type descriptor (the size depends only on the number of dimensions of the array, which is known by the compiler) and a pointer to the value, which is placed in the activation record after all static-sized local variables.

The central stack allows the compiler to generate efficient access code for the variables stored there in a statically scoped language. Let me abbreviate the phrase "accessed at a statically known offset" by the simpler but less precise "found." Static-sized local variables are found in the current activation record. Nonlocal variables are found in an activation record a certain distance from the front of the static chain; the compiler knows how many steps to take in that chain. Pointers to the values of dynamic-sized local variables are found in the current activation record; the values are interpreted according to type descriptors found either in the current record (if the type is declared locally) or in an activation record deeper on the static chain (for a nonlocal type).

The fourth region of runtime store, called the **heap**, is used for dynamic allocation of values accessed through pointers.[10] These values do not follow a stack discipline. This region of store expands as needed, getting increments from the operating system when necessary. To avoid ever-increasing store requirements for long-running programs, values are deallocated when they are no longer needed. The space can later be reallocated to new values. Deallocation can be triggered by explicit program requests (such as Pascal's `dispose` procedure) or by automatic methods such as reference counts and garbage collection. Reference counts indicate how many pointers are referencing each value. Each assignment and parameter binding modifies these counts, and each exit from a name scope reduces the counts for those variables that are disappearing. When a count is reduced to 0, the value may be deallocated and its space used for something else. Unfortunately, circular lists are never deallocated, even when they are no longer accessible. Garbage collection takes place when the store allocator notices that not much room is left. All accessible structures are recursively traversed and marked, and then all unmarked values are deallocated. The user often notices a distinct pause during garbage collection. There are incremental and concurrent garbage collection algorithms that reduce this interruption.

[10] Don't confuse the heap with the treelike data structure of the same name.

4 ◆ FINAL COMMENTS

This chapter has attempted to introduce the study of programming languages by placing it in the context of software tools in general. The background material on programming languages is, of necessity, very concise. Its aim is to lay the foundation for the concepts developed in the rest of this book.

The language concepts introduced here are in some sense the classical Algol-like structures. They are developed in various directions in the following chapters, each of which concentrates first on one programming language and then shows ideas from a few others to flesh out the breadth of the topic. Where appropriate, they end with a more mathematical treatment of the subject. Chapter 2 shows nonclassical control structures. Chapter 3 investigates the concept of data type. It presents a detailed discussion of ML, which shows how polymorphism can be incorporated in a statically typed language. Because ML is mostly a functional language, you may want to read Chapter 4 before the section on ML in Chapter 3.

The next chapters are devoted to nonclassical paradigms, that is, languages not descended from Algol. Chapter 4 discusses functional programming, concentrating on LISP. The concept of abstract data type is generalized in several ways in the next three chapters. Chapter 5 introduces object-oriented programming, concentrating on Smalltalk and C++. Chapter 6 discusses dataflow languages, concentrating on Val. Chapter 7 shows some of the wide range of development of languages for concurrent programming. A very different view of programming is presented in Chapter 8, which is is devoted to logic programming, concentrating on Prolog. Languages dealing with special-purpose data aggregates, such as strings, arrays, databases, and mathematical formulas, are discussed in Chapter 9. Finally, Chapter 10 shows several mathematical approaches to formalizing the syntax and semantics of programming languages; although it uses imperative languages as its model, such approaches have been used for the other language paradigms as well.

EXERCISES

Review Exercises

1.1 In what ways does C (or pick another language) fall short of the criteria in Section 2 for excellence?

1.2 How would you define the **mod** operator?

1.3 Show a code fragment in which short-circuit semantics for **or** yield a different result than complete-evaluation semantics.

1.4 Why do most languages with **case** statements prefer that the conditions have compile-time values?

1.5 Write a procedure that produces different results depending on whether its parameters are passed by value, reference, or name mode.

1.6 FORTRAN only passes parameters in reference mode. C only passes parameters in value mode. Pascal allows both modes. Show how you can get the effect of reference mode in C and how you can get the effect of value mode in FORTRAN by appropriate programming techniques. In particular, show in both FORTRAN and C how to get the effect of the following code.

Figure 1.23

```
variable X, Y : integer;

procedure Accept
     (A : reference integer; B: value integer);
begin
     A := B;
     B := B+1;
end; -- Accept

X := 1;
Y := 2;
Accept(X, Y);
-- at this point, X should be 2, and Y should be 2
```

1.7 If a language does not allow recursion (FORTRAN II, for example, did not), is there any need for a central stack?

1.8 C does not allow a procedure to be declared inside another procedure, but Pascal does allow nested procedure declarations. What effect does this choice have on runtime storage organization?

Challenge Exercises

1.9 Why are array slices usually allowed only in the last dimension?

1.10 Write a program that prints the index of the first all-zero row of an $n \times n$ integer matrix `M` [Rubin 88]. The program should access each element of the matrix at most once and should not access rows beyond the first all-zero row and columns within a row beyond the first non-zero element. It should have no variables except the matrix `M` and two loop indices `Row` and `Column`. The program may not use **goto**, but it may use multilevel **break** and **next**.

1.11 What is the meaning of a **goto** from a procedure when the target is outside the procedure?

1.12 Why do **goto** labels passed as parameters require closures?

1.13 Rewrite Figure 1.21 (page 21) so that procedure `A` takes a label instead of a procedure. The rewritten example should behave the same as Figure 1.21.

1.14 What rules would you make if you wanted to allow programmers to mix positional and keyword actual parameters?

1.15 The C language allows new name scopes to be introduced. However, C is not generally considered a block-structured language. Why not?

1.16 The text claims that the compiler knows the size of all global variables. Is this claim true for global dynamic-sized arrays?