

Language Tutorial

by K. N. King

TopSpeed Modula-2



9-1-89

Eric Streit

TopSpeedTM Modula-2

For IBM[®] Personal Computers and Compatibles

This manual was condensed from *Modula-2: A Complete Guide* (Prentice-Hall, 1989) and adapted to the TopSpeed Modula-2 system. The goal of the manual is to introduce the key features of Modula-2 quickly and painlessly. For a more detailed Modula-2, consult *Modula-2: A Complete Guide*, which contains detailed information on computer science with Pascal, shows how to use the language in programs and modules, covers a wide range of low-level programming, constraints, and interrupt handling, plus many other valuable features. (A coupon for ordering this book is enclosed; if the card is missing, use the coupon at the end of this tutorial.)

Language Tutorial

K. N. King

Much of Modula-2's power comes from its library of what is called software modules. There is no standard set of library modules for Modula-2 at present. Many Modula-2 systems provide the library modules described in *Programming in Modula-2*. TopSpeed Modula-2 includes a different (and much more extensive) library. The next pages in this tutorial illustrate the use of the TopSpeed library. For more information about the TopSpeed library, consult the *User's Manual*.

Organization

Chapter 1 provides an overview of Modula-2. The first part of this chapter explains the basic features of Modula-2 by comparison with Pascal. The second part of the chapter is a brief introduction to the TopSpeed Modula-2 system. Chapter 2 introduces the TopSpeed Modula-2 system. Chapter 3 introduces the TopSpeed Modula-2 system. Chapter 4 introduces the TopSpeed Modula-2 system. Chapter 5 introduces the TopSpeed Modula-2 system. Chapter 6 introduces the TopSpeed Modula-2 system. Chapter 7 introduces the TopSpeed Modula-2 system. Chapter 8 introduces the TopSpeed Modula-2 system. Chapter 9 introduces the TopSpeed Modula-2 system. Chapter 10 introduces the TopSpeed Modula-2 system. Chapter 11 introduces the TopSpeed Modula-2 system. Chapter 12 introduces the TopSpeed Modula-2 system. Chapter 13 introduces the TopSpeed Modula-2 system. Chapter 14 introduces the TopSpeed Modula-2 system. Chapter 15 introduces the TopSpeed Modula-2 system. Chapter 16 introduces the TopSpeed Modula-2 system. Chapter 17 introduces the TopSpeed Modula-2 system. Chapter 18 introduces the TopSpeed Modula-2 system. Chapter 19 introduces the TopSpeed Modula-2 system. Chapter 20 introduces the TopSpeed Modula-2 system. Chapter 21 introduces the TopSpeed Modula-2 system. Chapter 22 introduces the TopSpeed Modula-2 system. Chapter 23 introduces the TopSpeed Modula-2 system. Chapter 24 introduces the TopSpeed Modula-2 system. Chapter 25 introduces the TopSpeed Modula-2 system. Chapter 26 introduces the TopSpeed Modula-2 system. Chapter 27 introduces the TopSpeed Modula-2 system. Chapter 28 introduces the TopSpeed Modula-2 system. Chapter 29 introduces the TopSpeed Modula-2 system. Chapter 30 introduces the TopSpeed Modula-2 system. Chapter 31 introduces the TopSpeed Modula-2 system. Chapter 32 introduces the TopSpeed Modula-2 system. Chapter 33 introduces the TopSpeed Modula-2 system. Chapter 34 introduces the TopSpeed Modula-2 system. Chapter 35 introduces the TopSpeed Modula-2 system. Chapter 36 introduces the TopSpeed Modula-2 system. Chapter 37 introduces the TopSpeed Modula-2 system. Chapter 38 introduces the TopSpeed Modula-2 system. Chapter 39 introduces the TopSpeed Modula-2 system. Chapter 40 introduces the TopSpeed Modula-2 system. Chapter 41 introduces the TopSpeed Modula-2 system. Chapter 42 introduces the TopSpeed Modula-2 system. Chapter 43 introduces the TopSpeed Modula-2 system. Chapter 44 introduces the TopSpeed Modula-2 system. Chapter 45 introduces the TopSpeed Modula-2 system. Chapter 46 introduces the TopSpeed Modula-2 system. Chapter 47 introduces the TopSpeed Modula-2 system. Chapter 48 introduces the TopSpeed Modula-2 system. Chapter 49 introduces the TopSpeed Modula-2 system. Chapter 50 introduces the TopSpeed Modula-2 system. Chapter 51 introduces the TopSpeed Modula-2 system. Chapter 52 introduces the TopSpeed Modula-2 system. Chapter 53 introduces the TopSpeed Modula-2 system. Chapter 54 introduces the TopSpeed Modula-2 system. Chapter 55 introduces the TopSpeed Modula-2 system. Chapter 56 introduces the TopSpeed Modula-2 system. Chapter 57 introduces the TopSpeed Modula-2 system. Chapter 58 introduces the TopSpeed Modula-2 system. Chapter 59 introduces the TopSpeed Modula-2 system. Chapter 60 introduces the TopSpeed Modula-2 system. Chapter 61 introduces the TopSpeed Modula-2 system. Chapter 62 introduces the TopSpeed Modula-2 system. Chapter 63 introduces the TopSpeed Modula-2 system. Chapter 64 introduces the TopSpeed Modula-2 system. Chapter 65 introduces the TopSpeed Modula-2 system. Chapter 66 introduces the TopSpeed Modula-2 system. Chapter 67 introduces the TopSpeed Modula-2 system. Chapter 68 introduces the TopSpeed Modula-2 system. Chapter 69 introduces the TopSpeed Modula-2 system. Chapter 70 introduces the TopSpeed Modula-2 system. Chapter 71 introduces the TopSpeed Modula-2 system. Chapter 72 introduces the TopSpeed Modula-2 system. Chapter 73 introduces the TopSpeed Modula-2 system. Chapter 74 introduces the TopSpeed Modula-2 system. Chapter 75 introduces the TopSpeed Modula-2 system. Chapter 76 introduces the TopSpeed Modula-2 system. Chapter 77 introduces the TopSpeed Modula-2 system. Chapter 78 introduces the TopSpeed Modula-2 system. Chapter 79 introduces the TopSpeed Modula-2 system. Chapter 80 introduces the TopSpeed Modula-2 system. Chapter 81 introduces the TopSpeed Modula-2 system. Chapter 82 introduces the TopSpeed Modula-2 system. Chapter 83 introduces the TopSpeed Modula-2 system. Chapter 84 introduces the TopSpeed Modula-2 system. Chapter 85 introduces the TopSpeed Modula-2 system. Chapter 86 introduces the TopSpeed Modula-2 system. Chapter 87 introduces the TopSpeed Modula-2 system. Chapter 88 introduces the TopSpeed Modula-2 system. Chapter 89 introduces the TopSpeed Modula-2 system. Chapter 90 introduces the TopSpeed Modula-2 system. Chapter 91 introduces the TopSpeed Modula-2 system. Chapter 92 introduces the TopSpeed Modula-2 system. Chapter 93 introduces the TopSpeed Modula-2 system. Chapter 94 introduces the TopSpeed Modula-2 system. Chapter 95 introduces the TopSpeed Modula-2 system. Chapter 96 introduces the TopSpeed Modula-2 system. Chapter 97 introduces the TopSpeed Modula-2 system. Chapter 98 introduces the TopSpeed Modula-2 system. Chapter 99 introduces the TopSpeed Modula-2 system. Chapter 100 introduces the TopSpeed Modula-2 system.

Jensen & Partners International

TopSpeed™ is a trademark of Jensen & Partners International.

IBM® is a registered trademark of the International Business Machines Corporation.

Copyright © 1988 by Jensen & Partners International. All rights reserved.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Preface

This tutorial was condensed from *Modula-2: A Complete Guide* (Lexington, Mass.: D. C. Heath, 1988) and adapted to the JPI TopSpeed Modula-2 system. The goal of the tutorial is to introduce the key features of Modula-2 quickly and painlessly. For a closer look at Modula-2, consult *Modula-2: A Complete Guide*, which contains detailed, point-by-point comparisons with Pascal, more than 100 example programs and modules, extensive coverage of low-level programming, coroutines, and interrupt handling, plus numerous other valuable features. (A card for ordering this book is enclosed; if the card is missing, use the coupon at the end of this tutorial.)

Differences Between TopSpeed Modula-2 and Other Modula-2 Systems

At present, no national or international standard exists for Modula-2. The *de facto* standard for Modula-2 is Niklaus Wirth's *Programming in Modula-2*, Third, Corrected Edition (New York: Springer-Verlag, 1985). TopSpeed Modula-2 is based on the description of Modula-2 in this book. However, TopSpeed Modula-2 contains a number of useful extensions not present in the language as defined by Wirth.

Much of Modula-2's power comes from its libraries, where reusable software components reside. There is no standard set of library modules for Modula-2 at present. Most Modula-2 systems provide the library modules described in *Programming in Modula-2*. TopSpeed Modula-2 includes a different (and much more extensive) library. The programs in this tutorial illustrate the use of the TopSpeed library. For more information about the TopSpeed library, consult the *User's Manual*.

Organization

Chapter 1 provides an overview of Modula-2. The first part of this chapter explains the basic features of Modula-2 by comparison with Pascal. The second part covers Modula-2's advanced features, which have no counterpart in Pascal.

Chapters 2-5 cover the fundamentals of Modula-2. Chapter 2 describes Modula-2 both "bottom-up" (by explaining the rules for forming Modula-2 symbols) and "top-down" (by explaining the structure of a Modula-2 program). Chapter 3 presents Modula-2's numeric, character, and Boolean types and their use in forming expressions and declaring

constants and variables. Chapter 4 discusses Modula-2's assignment statement and control structures. Chapter 5 covers procedures (and functions).

Chapters 6–8 describe Modula-2's mechanisms for creating new types. Chapter 6 covers unstructured types (enumeration and subrange types), Chapter 7 explains structured types (arrays, records, and sets), and Chapter 8 discusses pointer and procedure types.

Chapter 9 describes modules, the most important new feature of Modula-2. This chapter covers both library modules and local modules; it also explains how modules can be separately compiled.

Chapters 10 and 11 cover the advanced features of Modula-2. Chapter 10 shows how to do low-level, machine-dependent programming in Modula-2. Chapter 11 describes Modula-2's support for coroutines and interrupt handling.

K. N. K.

Contents

1	An Overview of Modula-2	1
	Modula-2's Basic Features	2
	Example Program: Integer Exponentiation	3
	Symbols	4
	Declarations	5
	Types	5
	Expressions	8
	Statements	8
	Procedures	10
	Input/Output	10
	Modula-2's Advanced Features	11
	Modules	11
	Library Modules	12
	Opaque Types	14
	Separate Compilation	15
	Local Modules	16
	Low-Level Features	17
	Coroutines	18
	Interrupt Handling	18
2	The Structure of a Modula-2 Program	21
	The Vocabulary of Modula-2	21
	Identifiers	21
	Reserved Words	22
	Standard Identifiers	23
	Numbers	23
	Strings	24
	Special Symbols	25
	Comments	26
	Modula-2 Is Free-Format	27
	Program Modules	27
	Import Lists	28
	Blocks	30

Statements	30
Statement Sequences and the Empty Statement	31
3 Basic Types	33
Numeric Types	33
The INTEGER Type	34
The CARDINAL Type	35
The REAL Type	36
"Long" and "Short" Numeric Types	37
Mixed Expressions	37
The TopSpeed MATHLIB Module	38
Example Program: Converting a Fahrenheit Temperature to Celsius	39
The CHAR Type	39
Representing Nonprinting Characters	40
The ORD and CHR Functions	40
The CAP Function	41
Example Program: Encoding a Message	41
The BOOLEAN Type	42
Relational Operators	42
Logical Operators	43
Expressions	44
Constant Expressions	45
Constant and Variable Declarations	45
Constant Declarations	46
Variable Declarations	47
Example Program: Computing the Circumference and Area of a Circle	47
4 Statements	49
The Assignment Statement	50
The INC and DEC Procedures	51
The IF and CASE Statements	52
The IF Statement	52
The CASE Statement	54
The WHILE , REPEAT , and LOOP Statements	56
The WHILE Statement	56
The REPEAT Statement	57
The LOOP Statement	58
Example Program: Testing the MATHLIB Module	60
The FOR Statement	61
5 Procedures	65
Procedure Declarations	65
Procedure Calls	67

	Declarations of Local Entities	67
	The RETURN Statement	68
	Order of Procedure Declarations	69
	Parameters	70
	Variable Parameters	72
	Value Parameters	72
	Functions	73
6	Enumeration and Subrange Types	77
	Programmer-defined Types	77
	Type Specifications	78
	Type Declarations	79
	Enumeration Types	80
	The ORD and VAL Functions	82
	The INC and DEC Procedures	83
	Example: Writing Values of an Enumeration Type	84
	Subrange Types	84
	Compatibility and Assignment Compatibility	86
	Type Equivalence	86
	Compatibility	87
	Assignment Compatibility	88
	Summary	89
7	Array, Record, and Set Types	91
	Array Types	91
	Multidimensional Arrays	93
	Open Array Parameters	95
	String Types	96
	The TopSpeed Str Module	99
	Record Types	99
	Example: Updating Inventory Records	101
	Designators	101
	The WITH Statement	102
	Records with Variant Parts	103
	Set Types	106
	Constructing Set Values	108
	Set Assignment	108
	Set Operators	109
	Set Constants	109
	Relational Operators	110
	The INCL and EXCL Procedures	111
	Example: Checking Weekly Schedules	112

8	Pointer and Procedure Types	113
	Pointer Types	114
	Defining Pointer Types	114
	Creating Anonymous Variables	115
	Pointer Operations: Dereferencing, Assignment, and Comparison	116
	Disposing of Anonymous Variables	119
	Example: Using Pointers to Construct Linked Lists	120
	Procedure Types	123
	Defining Procedure Types	124
	Procedure Assignment	124
	Calling a Procedure Variable	125
	Procedures Used as Parameters	126
	Functions That Return Procedure Values	126
9	Modules	129
	Library Modules	130
	Definition Modules	133
	Example: The Definition Part of a Stack Module	134
	Implementation Modules	136
	Example: The Implementation Part of a Stack Module	138
	Order of Module Initialization	139
	Opaque Types	140
	Example: A Module That Exports an Opaque Stack Type	141
	Separate Compilation	144
	Compilation	145
	Linking	145
	Version Checking	145
	Local Modules	146
	Local Module Declarations	148
	Import into a Local Module	149
	Export from a Local Module	149
	Module Initialization	150
10	Low-Level Programming	153
	The BITSET Type	154
	Example Program: Sorting a File of CARDINAL Numbers	156
	The SYSTEM Module	158
	The WORD , BYTE , and LONGWORD Types	158
	The ADDRESS and SHORTADDR Types and the ADR Function	160
	Example: Copying a Block of Words	161
	Type Transfer Functions	161
	Example: Bitwise Operations on Words	163

Absolute Addresses	164
Example Program: Showing the Hardware Configuration of an IBM PC	165
11 Coroutines and Interrupt Handling	167
Coroutines	167
Coroutine Creation	169
Coroutine Transfer	171
Coroutine Termination	172
Example Program: Printing Fibonacci Numbers	172
Workspaces	174
Interrupt Handling	176
Module Priorities	177
Index	179

Chapter 1

An Overview of Modula-2

Designed by Niklaus Wirth of Pascal fame, Modula-2 has attracted wide attention since 1980, when a description of it was first published. From its obscure beginnings as the base language for Wirth's Lilith computer system, Modula-2 has gained widespread acceptance as an alternative to such languages as Pascal, C, and Ada.

It is difficult to discuss Modula-2 without mentioning Pascal, because many of Modula-2's features come directly from Pascal with little or no modification. Modula-2 retains the strengths of Pascal: strong typing, a rich variety of data types, and a powerful set of control statements. Like Pascal, Modula-2 is a simple language that is easy to learn, easy to compile, and capable of being translated into efficient object code.

Strong typing means that every variable and constant in a Modula-2 program has a fixed type that is checked during compilation to ensure that the item is used in a consistent manner. Consequently, many programming errors are detected during compilation, when they are easiest to locate and fix.

The wide selection of data structures—including arrays, records, and sets—that made Pascal such a popular language also appear in Modula-2. This variety makes the programmer's task easier: he or she can design data structures that closely reflect the real-world objects they model.

Pascal is famous for its control statements, which allow the construction of well-structured programs that contain few `goto` statements (often none). Modula-2 contains all of Pascal's structured control statements and more.

Wirth strongly believes that programming languages should be simple and easy to master. He has followed that principle in designing all of his languages, including Pascal and Modula-2. Wirth also believes that languages should allow efficient translation and execution. In designing Modula-2, Wirth included only features that he knew could be com-

piled into efficient object code. Many programming languages have been designed long before they were implemented; Modula-2, however, was developed at the same time that its first compiler was written, ensuring that hard-to-compile features would not creep into the language.

In addition to retaining the best features of Pascal, Modula-2 includes new features that make it applicable to a much broader range of applications than Pascal. These features include

- open array parameters
- procedure types
- opaque types
- modules
- separate compilation (with type-checking across compilation unit boundaries)
- features for low-level, machine-dependent programming
- coroutines
- interrupt handling

These new features make Modula-2 ideal for developing systems software: operating systems, compilers, editors, linkers, debuggers, and the like. But Modula-2's support for modular program development and object-oriented design make it suitable for writing almost any large software system. In addition, Modula-2's small size and efficient object code have made it popular for programming microcomputers.

This chapter provides a comprehensive overview of the language. The first section describes the basic features of Modula-2—the features that have Pascal counterparts—while the second explains the advanced features of the language. As we sketch the basic features of Modula-2, we will mention briefly how these features differ from Standard Pascal's. (*Note: Knowledge of Pascal is not a prerequisite for reading this tutorial. Readers who do not know Pascal may skip Chapter 1. For a detailed list of differences between Modula-2 and Pascal, see *Modula-2: A Complete Guide*.*)

Modula-2's Basic Features

This section sketches the basic features of Modula-2 and describes how they differ from the features of Standard Pascal.

In many respects, Modula-2 is a smaller language than Pascal. In particular, some of Pascal's standard features are provided in Modula-2 by library modules that are included as part of a Modula-2 system. For example, Modula-2 has no built-in procedures for input and output; these are supplied by library modules instead.

Example Program: Integer Exponentiation

Before discussing the features of Modula-2, we present a simple program, `ComputePower`, that gives the flavor of the language. `ComputePower` will accept two nonnegative integers, `i` and `j`, from the user, compute `i` to the power `j`, and display the result on the user's screen.

```

MODULE ComputePower;
  (* reads a pair of nonnegative integers i and j and prints
    the result of raising i to the power j *)

  FROM IO IMPORT RdCard, WrCard, WrStr, WrLn;

  VAR i, j, power: CARDINAL;

  PROCEDURE IntExp(base, exponent: CARDINAL): CARDINAL;
    (* returns "base" raised to the power "exponent" *)
    VAR i, result: CARDINAL;
  BEGIN
    result := 1;
    FOR i := 1 TO exponent DO
      result := result * base
    END;
    RETURN result
  END IntExp;

BEGIN
  WrStr("This program computes i to the power j.");
  WrLn;

  (* read values for i and j *)
  WrStr("Enter i: ");
  i := RdCard();
  WrStr("Enter j: ");
  j := RdCard();

  (* compute i to the power j *)
  power := IntExp(i, j);

  (* write the answer *)
  WrStr("i to the power j is: ");
  WrCard(power, 1);
  WrLn
END ComputePower.

```

At first glance, this program might appear to be written in Pascal. A closer look reveals some of the differences between Pascal and Modula-2:

- Modula-2 keywords, such as **BEGIN**, **END**, and **FOR**, must contain only uppercase letters. (In general, Modula-2 is sensitive to the case of letters.)
- A Modula-2 program begins with the reserved word **MODULE**, followed by the name of the program. This name must also appear at the end of the program, after the word **END**.
- Modula-2 supplies a new type, **CARDINAL**, whose values consist of the nonnegative integers (up to some limit).

- Modula-2 provides no standard procedures for input and output. Instead, I/O procedures are supplied by library modules that are furnished as part of a Modula-2 system. Because these procedures are not recognized by the compiler, they must be *imported* from a library module (**IO** in the sample program) before they can be used. Each input or output procedure in **IO** can read or write only a single value of a particular type. **WrCard**, for example, will write a single **CARDINAL** value; it can't be used to write two **CARDINAL** values or values of another type. The **WrLn** procedure writes the end-of-line character, marking the end of a line of output. The number 1 in **WrCard**(**power**, 1) is analogous to the "field width" of Pascal's **write** and **writeln** procedures: it indicates that the value of **power** should be written in a field of size 1 (or more if necessary). Specifying a value of 1 causes **power** to be written without leading blanks.
- In Pascal, the body of the **FOR** statement would be enclosed between the words **begin** and **end**, because it consists of several statements. Modula-2 instead adopts the rule that the body of a **FOR** statement is always terminated by the word **END** (even if it consists of a single statement) but never preceded by the word **BEGIN**. The same rule applies to all of Modula-2's other structured statements as well.

Symbols

Modula-2's symbols can be divided into the following categories:

- identifiers
- reserved words
- standard identifiers
- numbers (integer and real literals)
- strings
- special symbols

In most respects, Modula-2's symbols resemble Pascal's; there are only a few major differences:

- Modula-2 is case-sensitive: it distinguishes between upper- and lowercase letters in identifiers (thus, **i** and **I** are treated as two different identifiers). Reserved words must contain only uppercase letters. Pascal is not case-sensitive.
- Modula-2 has more reserved words than Pascal, because of added features that do not appear in Pascal. Yet Modula-2 has fewer standard identifiers, because many procedures that are standard in Pascal are supplied by library modules in Modula-2.
- Modula-2 allows octal and hexadecimal integer literals; for example, the decimal value 100 can also be expressed as **144B** (octal) or **64H** (hexadecimal). Pascal integers must be expressed in decimal form only. Modula-2 also allows the use of an octal number, followed by the letter **C**, to represent a character; for example, **12C** represents the ASCII line-feed character.

- In Modula-2, only the `(*` and `*)` symbols are allowed to delimit comments; comments can be nested. In Pascal, comments can be enclosed between either `{` and `}` or `(*` and `*)`; comments cannot be nested.

Declarations

Modula-2 allows the declaration of

- constants
- types
- variables
- procedures (including functions)
- modules

(TopSpeed Modula-2 also allows the declaration of labels, since it provides the `GOTO` statement as an extension to Modula-2.) Pascal allows the declaration of labels, constants, types, variables, and procedures (and functions).

Modula-2 allows the different kinds of declarations to appear in any order (constants need not be declared before types, for example). Furthermore, it is not necessary for all declarations of a particular kind to be grouped together; it is possible to declare several variables, then a few constants, then more variables, for example. Consequently, related declarations (of a type and variables of that type, say) may be grouped together. Pascal requires that declarations appear in a particular order: labels first, then constants, types, variables, and procedures. Like Pascal, Modula-2 forbids forward references in declarations (with the exception of pointer type declarations).

Types

Like Pascal, Modula-2 is a strongly typed language. Each variable and constant has a fixed type; when an operation is to be performed, its operands must have appropriate types. An attempt to perform an operation on operands of the wrong type (adding two characters, for example) will be detected by the compiler, resulting in an error message during compilation. Modula-2's rules for type-checking are even stricter than Pascal's. Integer and real numbers cannot be mixed in expressions—and integer values cannot be assigned to real variables—unless the programmer requests that an explicit conversion of type be performed. Because of its intended use as a systems programming language, Modula-2 allows type-checking to be disabled when necessary.

One of Modula-2's strengths is its rich variety of data types. Modula-2's types can be divided into four classes:

- unstructured types
 - numeric (`CARDINAL`, `INTEGER`, `LONGINT`, `REAL`, `LONGREAL`)

- character (**CHAR**)
- boolean (**BOOLEAN**)
- enumeration
- subrange
- structured types
 - array
 - record
 - set
- pointer types
- procedure types

Modula-2 provides nine *standard* (predefined) types: **CARDINAL**, **INTEGER**, **LONGINT**, **REAL**, **LONGREAL**, **CHAR**, **BOOLEAN**, **BITSET**, and **PROC**. The **CARDINAL**, **LONGINT**, **LONGREAL**, **BITSET**, and **PROC** types are new in Modula-2; the other standard types appear in Pascal. (TopSpeed Modula-2 also provides the types **SHORTCARD**, **LONGCARD**, and **SHORTINT** as language extensions. Several low-level types are also standard in TopSpeed Modula-2.)

An *enumeration type* is a type whose values are specified (enumerated) by the programmer and represented by identifiers:

```
TYPE Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

The other way to create a new unstructured type is by selecting a range of consecutive values from an existing unstructured type:

```
TYPE Weekdays = [Mon..Fri]; (* subrange of Days *)
```

Weekdays is called a *subrange type*. Modula-2's enumeration and subrange types are almost identical to Pascal's. (Note, however, that a Modula-2 subrange type must be enclosed in brackets.)

Modula-2 provides three structured types: arrays, records, and sets. An *array* is a structure whose components all have the same type and are accessed by position. For example, suppose that the type **Vector** is defined as follows:

```
TYPE Vector = ARRAY [1..10] OF REAL;
```

A variable **A** of type **Vector** is a structure with 10 components, each of type **REAL**, named **A[1]**, **A[2]**, ..., **A[10]**. (In Modula-2, unlike Pascal, the index type of an array is not enclosed in brackets. The index type of the array **A** is the subrange **[1..10]**; the brackets belong to the subrange, not the array specification.) Character strings are represented by arrays whose components are of type **CHAR**:

```
TYPE String10 = ARRAY [0..9] OF CHAR;
```


By convention, Modula-2 strings have 0 as the lower index bound (Pascal requires a lower bound of 1). Pascal's strings must be packed arrays; Modula-2 does not allow arrays (or records or sets, for that matter) to be declared **packed**.

A *record* is a collection of values, possibly of different types, whose components are accessed by name:

```
TYPE CharCount = RECORD
    ch: CHAR;
    count: CARDINAL
END;
```

A variable **rec** of type **CharCount** is a structure with two components, one of type **CHAR** and one of type **CARDINAL**. The first component is named **rec.ch** and the second **rec.count**.

The third structured type is the *set*. A set is an unordered collection of values of the same type, with no duplicates. When declaring a set type, the programmer must specify the universe from which the elements of the set are chosen:

```
TYPE DigitSet = SET OF [0..9];
```

Values of type **DigitSet** include **DigitSet{0, 2, 3, 9}**, **DigitSet{1, 4}**, and **DigitSet{}** (the empty set). (In Pascal, set values are enclosed between [and] and do not include the name of the set type.)

In addition to structured and unstructured types, Modula-2 provides *pointer types* and *procedure types*. A value of a pointer type is a "pointer" or "reference" to an anonymous variable that has been created during program execution. Each pointer type is "bound" to some other type—its values can point only at anonymous variables of that type. For example, if **NodePtr** is defined as

```
TYPE NodePtr = POINTER TO Node;
```

then a value of type **NodePtr** is a pointer to an anonymous variable of type **Node**. (In Pascal, the ^ symbol would be used instead of the words **POINTER TO**.)

Pascal allows procedures to be supplied as parameters to other procedures as though they were data values. Modula-2 takes this concept a step further—it allows procedures to be assigned to variables (including array and record components). A variable to which a procedure is assigned must have a procedure type, such as

```
TYPE Comparison = PROCEDURE (REAL, REAL): BOOLEAN;
```

Variables of type **Comparison** can be assigned functions that have two parameters, both of type **REAL**, and return a value of type **BOOLEAN**.

Expressions

Modula-2's operators are almost identical to Pascal's. These operators fall into four classes:

arithmetic:	+	-	*	/	DIV	MOD
logical:	NOT	~	AND	&	OR	
set:	+	-	*	/		
relational:	=	<>	#	<	<=	> >= IN

Three operators have alternate names: **NOT** can be written **~**, **AND** can be written **&**, and **<>** can be written **#** (their names in Pascal are **NOT**, **AND**, and **<>**). The set operation / (symmetric difference) does not appear in Pascal. TopSpeed Modula-2 also provides the operators **<<** (left shift) and **>>** (right shift) as a language extension.

Modula-2 performs "short-circuit" evaluation of Boolean expressions: a Boolean expression is evaluated from left to right only until the value of the expression can be determined. For example, the expression

```
(i <= length) & (A[i] # 0)
```

has the value **FALSE** if **i** is greater than **length**; the subexpression **(A[i] # 0)** is guaranteed not to be evaluated in this case. If **A**'s upper bound is **length**, short-circuit evaluation avoids an index-out-of-range error. Pascal does not guarantee short-circuit evaluation of Boolean expressions.

Modula-2 allows a constant expression wherever a constant value is expected:

```
CONST pi = 3.14159;
      twoPi = 2.0 * pi;
      oneK = 1024;
TYPE Buffer = ARRAY [0..3*oneK-1] OF CHAR;
```

Pascal does not follow this rule.

Mathematical functions (with the exception of **ABS**) are not standard in Modula-2; they must be imported from a library module.

Statements

Modula-2 provides a small but powerful set of eleven statements:

assignment	CASE	LOOP	RETURN
procedure call	WHILE	FOR	WITH
IF	REPEAT	EXIT	

(TopSpeed Modula-2 also provides the **GOTO** statement as a language extension.) Almost all these statements are for the control of program flow (only the assignment and

WITH statements do not affect the order in which statements are executed). Two statements (**IF** and **CASE**) allow a choice to be made among alternate control paths. The **WHILE**, **REPEAT**, **LOOP**, and **FOR** statements are used to repeatedly execute a group of statements. The **LOOP** statement allows a sequence of statements to be repeated indefinitely until an **EXIT** statement is executed. The **RETURN** statement allows a procedure to return before the end of its body is reached.

Modula-2's selection of control structures is so rich that virtually no situation requires a **GOTO** statement, a staple of many older programming languages. In addition to the **EXIT** and **RETURN** statements, which are essentially restricted **GOTO** statements, Modula-2 supplies a **HALT** procedure, which terminates program execution. (Pascal has nothing comparable to **EXIT**, **RETURN**, or **HALT**; the programmer must use **goto** statements instead.)

Modula-2's statements differ from Pascal's in several ways:

- Modula-2's **CASE** statement, unlike Pascal's, allows an **ELSE** clause, which is executed if the value of the case expression matches none of the case labels. Modula-2's **CASE** statement also allows ranges bounded by constant expressions as case labels.
- Pascal has no statement corresponding to Modula-2's **LOOP** statement. (The same effect can be achieved in other ways, however. For example, a **LOOP** statement can be simulated by a **while** statement that tests a condition that is always **true**, with **goto** statements instead of **EXIT** statements.)
- Modula-2's **FOR** statement includes an optional **BY** clause that allows the programmer to specify the step size as a constant expression; Pascal's **for** statement allows only **+1** (**to**) and **-1** (**downto**) as step sizes.

Modula-2's **IF**, **WHILE**, **FOR**, and **WITH** statements may contain statement sequences and always end with the reserved word **END**. Pascal's **if**, **while**, **for**, and **with** statements may contain only single statements, which may be compound (a compound statement is a sequence of statements bracketed by **begin** and **end**). For example, the body of a Modula-2 **WHILE** statement is a sequence of statements, followed by the word **END**; the body of a Pascal **while** statement is a single statement, which may be compound:

Modula-2

```
WHILE Expression DO
  StatementSequence
END
```

Pascal

```
while Expression do while Expression do
  Statement          begin
                    StatementSequence
                    end
```

The net effect is that the body of a Modula-2 **WHILE** statement is always followed by the word **END**, but never preceded by the word **BEGIN**.

Procedures

Modula-2 allows the declaration of procedures (sometimes called *subprograms* or *subroutines* in other programming languages). In Modula-2, the term *procedure* includes functions (called *function procedures* in Modula-2) since declarations of both begin with the word **PROCEDURE**. (In Pascal, function declarations begin with the word **function**.) In Modula-2, a function returns a value by executing a **RETURN** statement that specifies the desired value. In Pascal, the value to be returned by a function is assigned to the function name as though it were a variable.

Like Pascal, Modula-2 offers two kinds of parameters: variable parameters and value parameters. Changes made to a variable parameter from within a procedure cause the corresponding actual parameter to change; changes made to a value parameter do not affect the actual parameter.

In most respects, Modula-2 procedures are virtually identical to Pascal procedures. However, Modula-2 has one important feature that is missing in Level 0 Standard Pascal: the *open array parameter*. For example, if **P** is declared as

```
PROCEDURE P(x: ARRAY OF REAL);
```

then the actual parameter supplied when **P** is called can be any one-dimensional array of **REAL** values. Level 1 Standard Pascal contains a feature (conformant array parameters) similar to Modula-2's open array parameters.

Input/Output

The Modula-2 language provides no input/output procedures. Instead, programs must use I/O procedures that are supplied by library modules. In TopSpeed Modula-2, these modules are named **IO** and **FIO**. Programmers may also write their own I/O libraries.

I/O procedures supplied by a library module are subject to the same rules as all Modula-2 procedures. In particular, each I/O procedure must have a fixed number of parameters and each parameter must have a fixed type. Consequently, I/O procedures normally allow only one value to be read or written at a time, as we saw in the sample program:

```
WrStr("i to the power j is: ");
WrCard(power, 1);
WrLn
```

Pascal's I/O procedures allow varying numbers of parameters of varying types; in Pascal, these three calls would be replaced by one call of **writeln**:

```
writeln("i to the power j is: ", power: 1);
```


Modula-2 has no special syntax for indicating to an output procedure how many characters should be written (the "field width," in Pascal terms); this value must be supplied to the procedure as a parameter (as in the call of `WrCard`).

Unlike Pascal, the Modula-2 language does not provide `file` types. However, Modula-2 systems normally include a file-handling module (`FIO` in TopSpeed Modula-2) that supplies a file type and a collection of procedures that perform file operations.

Modula-2's Advanced Features

Although Modula-2 has much in common with Pascal, the former provides a number of important features not present in the latter. These features can be divided into the following categories:

- modules
- opaque types
- separate compilation
- low-level features
- coroutines
- interrupt handling

Modules, opaque types, and separate compilation are essential features of Modula-2 with which every programmer must become familiar. Low-level features, coroutines, and interrupt handling are used primarily in systems programming.

Modules

The primary difference between Modula-2 and Pascal is the former's support for modules. In Pascal, a program is a single entity that is compiled as a whole (although many Pascal compilers allow some form of separate compilation). In Modula-2, however, a program consists of one or more modules. One of these (the *program module*) contains the main program; the others are called *library modules*.

Library modules serve as convenient repositories of related entities (constants, types, variables, and procedures). Not all of the entities defined in a module are available for use by other modules, however. Modules act as walls: Entities declared inside a module are not visible outside the module unless the module voluntarily *exports* them. In order for another module to gain access to any of these exported entities, the module must *import* the entity.

A program module may import from any number of library modules; these modules, in turn, may import from other library modules. Thus, a complete Modula-2 program consists of a set of interconnected modules, with one—the program module—serving as the main program. (In addition, modules may be declared inside other modules; a module

nested inside another is called a *local module*.) Programs consisting of a single module are rare; most programs import input/output procedures from library modules, because the Modula-2 language provides no standard I/O procedures.

Library Modules

Library modules have two parts: a *definition part* and an *implementation part*. The definition part specifies the entities that the module exports; the implementation part gives the rest of the module. For example, if a procedure is exported by a library module, only the procedure heading appears in the definition part of the module; the implementation part contains the full declaration of the procedure. In other words, the definition part describes *what* the module does; the implementation part describes *how* it does it.

Consider the program **ComputePower** that appeared earlier in this chapter. This program makes use of the (function) procedure **IntExp**, which returns *i* to the power *j* when given two **CARDINAL** numbers *i* and *j*. This procedure will be needed by many programs, because Modula-2 does not provide an exponentiation operator. To avoid having to redefine **IntExp** in these other programs, we could create a library module called **ExpLib** that would export **IntExp**; the **ComputePower** program could then import **IntExp** from **ExpLib**. We might wish to make **ExpLib** more useful by including other procedures besides **IntExp**. A logical choice would be **RealExp**, a procedure that computes *x* to the power *y* for **LONGREAL** values *x* and *y*. The definition part of **ExpLib** might then look like this:

```
DEFINITION MODULE ExpLib;
  (* exports exponentiation functions *)

  PROCEDURE IntExp(base, exponent: CARDINAL): CARDINAL;
    (* returns "base" raised to the power "exponent" *)

  PROCEDURE RealExp(base, exponent: LONGREAL): LONGREAL;
    (* returns "base" raised to the power "exponent";
       terminates program if base is not positive *)

END ExpLib.
```

The definition part of **ExpLib** includes all the information that a programmer needs in order to use the procedures **IntExp** and **RealExp**. The procedure headings show how many parameters are expected, what their types are, and what type of result is returned. The comments that accompany each procedure indicate what it does and discuss any special constraints on the procedure's parameters. Information not needed in order to call the procedures (such as the algorithms used to perform exponentiation) is deliberately omitted from the definition part of the module.

Using just the definition part of **ExpLib**, we can rewrite **ComputePower** to import **IntExp** instead of declaring it. **ComputePower** will import **IntExp** from **ExpLib**

in the same way that it imports procedures from IO. The revised version of `ComputePower` would look like this:

```
MODULE ComputePower;
  (* reads a pair of nonnegative integers i and j and prints
     the result of raising i to the power j *)

  FROM IO IMPORT RdCard, WrCard, WrStr, WrLn;
  FROM ExpLib IMPORT IntExp;

  VAR i, j, power: CARDINAL;

BEGIN
  WrStr("This program computes i to the power j.");
  WrLn;

  (* read values for i and j *)
  WrStr("Enter i: ");
  i := RdCard();
  WrStr("Enter j: ");
  j := RdCard();

  (* compute i to the power j *)
  power := IntExp(i, j);

  (* write the answer *)
  WrStr("i to the power j is: ");
  WrCard(power, 1);
  WrLn
END ComputePower.
```

The implementation part of a library module gives complete information about the entities listed in the definition part. When a procedure heading appears in the definition part, the implementation part must contain the full procedure declaration, including the heading (which must match the heading in the definition part). Because the definition part of `ExpLib` contains headings for `IntExp` and `RealExp`, the implementation part must include full declarations of `IntExp` and `RealExp`. The declaration of `IntExp` can be copied from the first version of `ComputePower`. To implement the procedure `RealExp`, we can use the formula

$$a^b = e^{b \ln(a)}$$

provided that $a > 0$. The implementation part of `ExpLib` follows:

```
IMPLEMENTATION MODULE ExpLib;

  FROM IO IMPORT WrStr, WrLn;
  FROM MATHLIB IMPORT Exp, Log;

  PROCEDURE IntExp(base, exponent: CARDINAL): CARDINAL;
    (* returns "base" raised to the power "exponent" *)
    VAR i, result: CARDINAL;
  BEGIN
    result := 1;
    FOR i := 1 TO exponent DO
      result := result * base
    END;
  END;
```



```

    RETURN result
END IntExp;

PROCEDURE RealExp(base, exponent: LONGREAL): LONGREAL;
  (* returns "base" raised to the power "exponent";
     terminates program if base is not positive *)
BEGIN
  IF base <= 0.0 THEN
    WrStr("Error: RealExp called with illegal base");
    WrLn;
    HALT
  END;
  RETURN Exp(exponent * Log(base))
END RealExp;

END ExpLib.

```

The mathematical functions **Exp** and **Log** are not standard in Modula-2, but must be imported from a library module (**MATHLIB** in TopSpeed Modula-2).

We have seen one of the advantages of declaring **IntExp** in a library module rather than in **ComputePower**: **IntExp** can be shared with other programs. Another advantage is the ease with which **IntExp** can be modified. "Real-world" programs undergo frequent changes, not only to correct errors, but also to add new features or improve performance. When a program is divided into modules, many modifications affect only the implementation part of a single library module.

Opaque Types

The **ExpLib** module illustrates the usefulness of library modules for collecting frequently used procedures. Library modules have many other uses, as well. One of the most important features of Modula-2 is its provision for hiding the structure of a data type within a module. A type whose structure is hidden in this fashion is called an *opaque type*. The definition of an opaque type in the definition part of a library module is an abbreviated form of a normal type declaration; it consists only of the reserved word **TYPE** and the name of the type (the rest of the declaration is omitted).

For example, suppose that we are writing a program that requires a stack of characters. As we will see later, it can be advantageous to declare a stack type (which we will call **CharStack**) and hide the structure of this type in a library module. This module (call it **CharacterStacks**) must export procedures for manipulating variables of type **CharStack**, because no other module will know what the **CharStack** type really looks like. Here is the definition part of **CharacterStacks**:

```

DEFINITION MODULE CharacterStacks;
  (* exports type CharStack (stack of characters) and
     procedures for manipulating variables of type CharStack *)

  TYPE CharStack;    (* opaque export *)

  PROCEDURE Initialize(VAR s: CharStack);
    (* creates a stack s and makes it empty *)

```



```

PROCEDURE Push(VAR s: CharStack; ch: CHAR);
  (* pushes ch onto stack s; terminates program if stack is full *)

PROCEDURE Pop(VAR s: CharStack; VAR ch: CHAR);
  (* stores top element of stack s into ch, then pops s;
   terminates program if stack is empty *)

PROCEDURE IsEmpty(s: CharStack): BOOLEAN;
  (* returns TRUE if s is empty, FALSE otherwise *)

END CharacterStacks.

```

The definition part of **CharacterStacks** gives no information about **CharStack** other than its name. Modules that import the **CharStack** type can use it to declare variables, but the only way to manipulate these variables is by calling the procedures **Initialize**, **Push**, **Pop**, and **IsEmpty**. Modula-2 requires that a full declaration of **CharStack** appear in the implementation part of **CharacterStacks**, along with full declarations of **Initialize**, **Push**, **Pop**, and **IsEmpty**. (The details are beyond the scope of this chapter; the implementation part of **CharacterStacks** appears in Chapter 9.)

Separate Compilation

Although separate compilation is not a required feature of Modula-2, virtually every Modula-2 system provides it. Program modules and the definition and implementation parts of library modules can be separately compiled, and hence are called *compilation units*. To emphasize that the definition part of a library module can be compiled separately from the implementation part, the former is called a *definition module* and the latter an *implementation module*.

Once the modules in a program have been compiled, they must be linked together before execution. Some Modula-2 systems require an explicit linking phase; others perform the linking automatically when the user requests that a program be executed. (TopSpeed Modula-2 requires a linking phase.)

For the time being, we will restrict our attention to programs that do not include any programmer-defined library modules. For these programs, it is only necessary to compile the program module and then link it with any system-provided library modules (such as **IO** or **MATHLIB**) that it may need. When a program is linked, TopSpeed Modula-2 does not require the programmer to name all the modules to be linked. Instead, it asks only for the name of the program module; the linker then determines which modules need to be linked to the program module. These modules will include those from which the program module imported entities, modules from which those modules imported, and so forth. (For example, a request to link **ComputePower** will cause **ExpLib** to be linked also, because **ComputePower** imports from **ExpLib**. But **ExpLib** imports from **IO** and **MATHLIB**, so these modules must also be linked. **IO** and **MATHLIB** may import from still other modules, and so the process continues.)

Local Modules

Because local modules are declared inside compilation units, they cannot be separately compiled. Local modules are used for one purpose: to control the visibility of names within a program. A local module acts like a wall around the entities inside it. Entities visible outside the module are not visible inside it unless explicitly imported by the module; by the same token, entities visible inside the module are not visible outside unless exported by the module.

Consider the following program, named **PrintFibonacciNumbers**. This program prints the numbers in the sequence

1, 1, 2, 3, 5, 8, 13, ...

up to some predetermined limit. The Fibonacci numbers are generated by the **NextFibonacci** function. Each time it is called, **NextFibonacci** returns the next number in the sequence. In order to keep track of which number should be generated next, **NextFibonacci** needs two variables **fib1** and **fib2**. Because these variables are used only by **NextFibonacci**, they need not be visible throughout the rest of the program. Yet **fib1** and **fib2** cannot be declared local to **NextFibonacci**, because their values would be lost between calls of **NextFibonacci**. Declaring **fib1** and **fib2** in a local module along with **NextFibonacci**, but exporting only **NextFibonacci**, guarantees that **fib1** and **fib2** cannot be seen elsewhere in the program.

```
MODULE PrintFibonacciNumbers;
  (* prints Fibonacci numbers *)

  FROM IO IMPORT WrCard, WrLn;

  CONST maxFib = 10000;    (* prints numbers up to this limit *)

  VAR fib: CARDINAL;

  MODULE FibonacciGenerator;
    (* exports NextFibonacci while hiding fib1 and fib2 *)

    EXPORT NextFibonacci;

    VAR fib1, fib2: CARDINAL;

    PROCEDURE NextFibonacci(): CARDINAL;
      (* returns the next Fibonacci number, in order, each time
         it is called; the first two calls return 1 *)
    BEGIN
      IF fib1 < fib2 THEN
        INC(fib1, fib2);
        RETURN fib2
      ELSE
        INC(fib2, fib1);
        RETURN fib1
      END
    END NextFibonacci;
```



```

BEGIN
  fib1 := 0;
  fib2 := 1
END FibonacciGenerator;

BEGIN
  LOOP
    fib := NextFibonacci();
    IF fib > maxFib THEN EXIT END;
    WrCard(fib, 1);
    WrLn
  END
END PrintFibonacciNumbers.

```

INC is a standard procedure in Modula-2; INC(*i*, *n*) is equivalent to the statement *i* := *i* + *n*.

Low-Level Features

In some applications, a program must be able to perform low-level, machine-dependent operations (examining the address of a variable or changing the contents of a machine register, for example)—operations that are normally not possible when using a high-level programming language. Modula-2 was designed for systems programming, so it provides special features for performing low-level operations.

The data type **BITSET** allows ready access to the individual bits of a word. For example, the **BITSET** value {0, 5, 8, 15} represents a word containing 1's in positions 0, 5, 8, and 15, and 0's in all other positions. By applying combinations of normal set operations to **BITSET** values, it is possible to perform any desired bit operation on a word.

The module **SYSTEM** is another source of support for low-level programming. **SYSTEM** supplies the programmer with system-dependent data types, such as **ADDRESS** and **WORD**, that have special properties. **SYSTEM** also provides procedures such as **ADR**, which allows a program to determine the run-time address of a variable:

```

VAR addr: ADDRESS;
.
.
.
addr := ADR(x);    (* fetch address of variable x *)

```

(**ADDRESS**, **WORD**, and **ADR** are standard in TopSpeed Modula-2 and need not be imported from **SYSTEM**.)

Type transfer functions provide a way to circumvent Modula-2's normal type-checking mechanism. For example, if *i* is an **INTEGER** variable and *b* is a **BITSET** variable, the assignment

```
b := i;
```


is forbidden, because **b** and **i** have different types. However, using the type transfer function **BITSET**, we can write

```
b := BITSET(i);
```

The advantage of viewing **i** as a **BITSET** value is that its individual bits are now directly accessible. Type transfer can be used between any pair of types, either standard or programmer-defined, provided that the values of the two types occupy the same amount of storage.

Finally, many Modula-2 systems provide a mechanism for specifying the run-time address of a variable. In TopSpeed Modula-2, an address is a *segment:offset* pair. The declaration

```
VAR equipmentFlag [0:410H]: BITSET;
```

tells the compiler to locate the variable **equipmentFlag** at the address whose segment is 0 and whose offset is 410 (hex). The word located at this address can then be accessed through **equipmentFlag**.

Coroutines

Coroutines are similar to procedures. However, coroutines do not use the call-and-return mechanism that procedures do. A coroutine begins execution when control is transferred to it; it ceases to execute when it transfers control to another coroutine. Should control ever be transferred to the coroutine again, it will not begin executing from the beginning, but will instead resume from where it left off. Thus, the state of a coroutine's computation is preserved when the coroutine is not executing.

Coroutines are particularly useful in the construction of operating systems and other programs in which processes must appear to operate concurrently. By switching among several coroutines at frequent intervals, a program can make it appear that the coroutines are executing simultaneously.

Modula-2 allows a program to create coroutines (by calling the procedure **NEWPROCESS**) and switch between them (by calling **TRANSFER**). Both **NEWPROCESS** and **TRANSFER** must be imported; neither is predefined.

Interrupt Handling

Systems programmers must often be concerned with interrupts (hardware-generated signals indicating that a particular event has occurred—the completion of an input/output operation, for example). Modula-2's support for coroutines allows the language to provide a simple method of writing interrupt handlers (routines that respond to interrupts). The programmer writes the interrupt handler as a coroutine, then causes the coroutine to

be suspended until the desired interrupt occurs. (The suspension of the interrupt handler is done by a call of **IOTRANSFER**, which must be imported.) When the interrupt occurs, control is transferred to the interrupt handler, which then performs whatever processing is needed to respond to the interrupt. The handler then suspends itself until the next interrupt.

At times it is important to disable all or selected interrupts (for example, an interrupt handler should not be interrupted by the interrupt being serviced). Modula-2 provides a simple mechanism for inhibiting interrupts: the presence of a *priority* in the heading of a module indicates which interrupts are to be disabled when the module is being executed.

Chapter 1 presented an overview of Modula-2. This chapter begins a more detailed description of the language, starting with the symbols that make up a Modula-2 program and concluding with the overall structure of a Modula-2 main program.

The Vocabulary of Modula-2

We begin our exploration of Modula-2 by explaining the rules for forming the symbols that can appear in Modula-2 programs. These symbols include:

- identifiers
- reserved words
- stringed identifiers
- numbers (integers and real)
- strings
- special symbols

We will also show how to include comments in a program.

Identifiers

Programmer-defined names are called *identifiers* in Modula-2. An identifier is a letter followed by zero or more letters and digits. (In TopSpeed Modula-2, identifiers may include underscore, as well; this is an extension to Modula-2.)

The following are all valid identifiers:

```
x  alpha  testNumber  DONE  String10
```


Chapter 2

The Structure of a Modula-2 Program

Chapter 1 presented an overview of Modula-2. This chapter begins a more detailed description of the language, starting with the symbols that make up a Modula-2 program and continuing with the overall structure of a Modula-2 main program.

The Vocabulary of Modula-2

We begin our exploration of Modula-2 by explaining the rules for forming the symbols that can appear in Modula-2 programs. These symbols include

- identifiers
- reserved words
- standard identifiers
- numbers (integer and real)
- strings
- special symbols

We will also show how to include comments in a program.

Identifiers

Programmer-defined names are called *identifiers* in Modula-2. An identifier is a letter followed by zero or more letters and digits. (In TopSpeed Modula-2, identifiers may include underscores as well; this is an extension to Modula-2.)

The following are all valid identifiers:

```
x  oldch  SortNumbers  DONE  String10
```


The following are *not* valid identifiers:

```
Get!Next!Character <> 30X
```

The first two contain characters other than letters and digits; the third begins with a digit.

Modula-2 is a *case-sensitive* language: Two identifiers are the same only if both contain exactly the same characters in the same order; the uppercase and lowercase versions of a letter are not considered the same character. Thus, the following identifiers are all different:

```
oldch OLDCH OldCh OLDch oldCH oLdCh
```

It is not a good idea for a program to contain two identifiers that are the same except for capitalization; a person reading the program (not to mention the original programmer) is likely to confuse the two.

Modula-2 imposes no limit on the length of an identifier; all characters in an identifier are significant.

Reserved Words

Modula-2 does not allow programmers to use the following symbols (called *reserved words*) as identifiers:

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	SET
BY	EXPORT	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMENTATION	PROCEDURE	VAR
DO	IMPORT	QUALIFIED	WHILE
ELSE	IN	RECORD	WITH

Six of these (AND, DIV, IN, MOD, NOT, and OR) are operators; the others are keywords that help Modula-2 compilers recognize language constructs. Reserved words must contain only uppercase letters. (TopSpeed Modula-2 contains three additional reserved words: FORWARD, GOTO, and LABEL.)

Standard Identifiers

The following identifiers, called *standard identifiers*, have predefined meanings:

ABS	CHAR	FLOAT	INTEGER	NIL	SIZE
BITSET	CHR	HALT	LONGINT	ODD	TRUE
BOOLEAN	DEC	HIGH	LONGREAL	ORD	TRUNC
CAP	EXCL	INC	MAX	PROC	VAL
CARDINAL	FALSE	INCL	MIN	REAL	

These identifiers represent standard types (**CARDINAL**), standard constants (**FALSE**), and standard procedures (**ODD**). (In TopSpeed Modula-2, the following identifiers are also standard: **ADDRESS**, **ADR**, **BYTE**, **DISPOSE**, **LONGCARD**, **LONGWORD**, **NEW**, **NULLPROC**, **SHORTADDR**, **SHORTCARD**, **SHORTINT**, **VSIZE**, **WORD**.)

Do not confuse standard identifiers with reserved words. A program can declare a standard identifier to have a different meaning; reserved words cannot be redefined. Redefining a standard identifier is usually a poor idea, because this prevents using the identifier with its standard meaning and confuses anyone who is reading the program.

Standard identifiers, like reserved words, must contain only uppercase letters. Consequently, these symbols stand out in a program, making the structure of the program more apparent. Because all reserved words and standard identifiers in Modula-2 contain only capital letters, it is good practice to use lowercase or mixed uppercase and lowercase letters for programmer-defined identifiers. In identifiers that consist of more than one word, such as **TestForBlank**, it is good style to begin each word (except possibly for the first) with an uppercase letter.

Numbers

Two kinds of numbers (that is, numeric literals) may appear in Modula-2 programs: integers and real numbers. An integer is a sequence of digits:

0 231 32767

In Modula-2, integers are expressed in base 10, by default; the language also allows octal (base 8) and hexadecimal (base 16) literals. An octal integer is a sequence of octal digits (digits between 0 and 7) followed by the letter **B**:

0B 347B 77777B

(The letter **B** is intended to suggest the numeral 8.) A hexadecimal integer is a sequence of hexadecimal digits (digits between 0 and 9, plus the letters between **A** and **F**) followed by the letter **H**:

0H 0E7H 7FFFH

A hexadecimal number must always begin with a decimal digit. Without this rule, a hex number beginning with a letter (**E7H**, for example) would appear to be an identifier. Fortunately, adding the digit 0 to the beginning of the number (**0E7H**, in our example) will make it legal.

A real number must contain a decimal point preceded by at least one digit. No digits need follow the decimal point. The following are legal real numbers in Modula-2:

```
143.87  0.253  3902.
```

A real number may include a *scale factor*. A scale factor consists of the letter **E** (uppercase only), an optional sign, then one or more digits:

```
1.2944E10  3.E-8  15.0E+12
```

A number's scale factor indicates the power of 10 by which the number should be scaled. For example, **1.2944E9** (1.2944×10^9 in scientific notation) represents the number 1,294,400,000.

Strings

A *string* in Modula-2 is a sequence of characters enclosed between single or double quote marks:

```
'error 14'
"error 14"
```

Single and double quotes are not interchangeable; if a string begins with a single quote, it must end with a single quote (the same is true for double quotes).

If double quotes delimit a string, then the single quote mark may appear within the string. If single quotes delimit the string, then double quotes can appear inside the string:

```
"Madam, I'm Adam"
'Spell the word "obfuscation"'
```

A string cannot contain both single and double quotes:

```
"This string's not "legal""
```

Single-character strings obey the same rules as do longer strings. For example, both **'A'** and **"A"** represent the letter A.

Modula-2 provides no way to include nonprinting characters in a string. (However, the language does offer a special notation for representing a single nonprinting character; see Chapter 3.) Strings, like other symbols, cannot be split over two lines.

As a language extension, TopSpeed Modula-2 allows the use of the + operator to concatenate (literal) strings and/or single characters into a single long string. This operator can be used to overcome Modula-2's restriction that strings cannot be split over lines. Whenever we have a string that is too long to fit on a line, we break it into smaller strings, which we then join using the + operator:

```
"This string has been divided " +
"into three parts that are " +
"joined by the + operator"
```

The + operator also allows us to include nonprinting characters in a string.

Special Symbols

Modula-2 uses the following special symbols:

+	addition, set union
-	subtraction, set difference
*	multiplication, set intersection
/	real division, symmetric set difference
&	logical conjunction
~	logical negation
=	equal
# <>	not equal
<	less than
<=	less than or equal to, set inclusion
>	greater than
>=	greater than or equal to, set inclusion
:=	assignment
()	expressions; enumeration types; parameter lists
{ }	sets
[]	subrange types; array indexing; module priorities
(**)	comments
^	pointer dereferencing
' "	strings
, ; : . . .	punctuation

TopSpeed Modula-2 also recognizes << and >> as special symbols; they represent the left shift and right shift operators, respectively. In TopSpeed Modula-2, the + symbol has an additional meaning (concatenation of literal strings).

Comments

Any sequence of characters between the symbols `(*` and `*)` is a *comment*.

```
(* This is an example of a comment *)
```

Comments can appear anywhere in a program, either within a line or extending over many lines. Although comments usually occur between declarations or statements, they may appear between any two symbols:

```
CONST pi = (* to five decimal places *) 3.14159;
```

Comments may not divide a symbol, however:

```
CONST pi = 3.1415(* unsure about this digit *)9;    (* illegal*)
```

In general, comments are ignored (treated as blanks); they serve only to document a program. In TopSpeed Modula-2, however, comments may contain compiler directives. For example, the following comments signal the TopSpeed compiler to disable the test for arithmetic overflow for the duration of one statement:

```
(* $O-*)
i := j * k;
(* $O+*)
```

Modula-2 allows the programmer to nest comments. This ability is useful to temporarily take some portion of a program out of commission, perhaps during testing:

```
(*****
WrStr("The value of i is currently ");
WrCard(i, 1); (* i should have the value 10 at this point *)
WrLn;
*****)
```

The symbol `*)` after the word `point` ends only the comment begun on that line, not the comment begun two lines earlier. All the statements shown are within a comment, so none can be executed. (The lines of asterisks are to help readers locate the beginning and end of the comment, which may be many lines apart.)

The ability to nest comments provides a second benefit. If a programmer accidentally omits the closing symbol `*)` somewhere in a program, at the end of the program the compiler will detect a badly formed comment and will advise the programmer. In a language that does not allow nested comments, an omitted closing symbol will—without any indication to the programmer—cause the compiler to treat as a comment the portion of the program up to the next closing symbol.

Modula-2 Is Free-Format

Modula-2 programs are "free-format": they can be spread out over many lines or compressed into a few lines. Modula-2 compilers ignore blanks and ends of lines (allowing the programmer freedom to format the program for easier reading)—with two exceptions. First, a symbol cannot be split between two lines or divided by a blank or a comment. Second, the programmer must separate identifiers, reserved words, and numbers by at least one blank whenever ambiguity would result. In the statement

```
IF i > max THEN max := i END;
```

the blanks between **IF** and **i**, **max** and **THEN**, **THEN** and **max**, and **i** and **END** are all mandatory. If no blank appeared between **IF** and **i**, for example, a Modula-2 compiler would see a single identifier, **IFi**. The other blanks, such as those around the **>** operator, are not required.

Program Modules

A Modula-2 program consists of a *program module* and possibly one or more *library modules*. The program module serves as the "main program," while the library modules provide services to the program module (or to each other). For example, consider the following program, which writes the digits of a nonnegative integer in reverse order:

```
MODULE ReverseNumber;
  (* writes the digits of a nonnegative integer in reverse order *)

  FROM IO IMPORT RdCard, WrCard, WrStr, WrLn;

  VAR number: CARDINAL;

BEGIN
  WrStr("Enter number to be reversed: ");
  number := RdCard();
  WrLn;

  WrStr("The reversal is: ");
  REPEAT
    WrCard(number MOD 10, 1);
    number := number DIV 10
  UNTIL number = 0;
  WrLn
END ReverseNumber.
```

This module, together with the **IO** library module (and any modules from which **IO** imports) comprises a complete Modula-2 program. In order to write small programs in Modula-2, we need only know the rules for writing program modules and have some knowledge of I/O library modules. To write large programs, however, we need to know how to write library modules (Chapter 9).

A program module begins with the reserved word **MODULE**, followed by an identifier (the program name). A *priority* may optionally follow the program name, specifying an interrupt priority for the module. (We discuss interrupt handling in Chapter 11; until then, we always assume that the priority is omitted.) Next comes a semicolon, followed by the import lists—lists of entities to be imported from library modules. (Although no import lists are necessary, most programs import input/output procedures from a library module.) After the import lists appears a block containing the module's declarations and statements, then the program name, and finally a period. The name at the end of a program module must be identical to the name that appears at the beginning; even the case of corresponding letters must be the same.

Notice that Modula-2 makes no provision for a program module to export entities—program modules are "consumers" of services provided by library modules, never "producers."

Having covered the overall structure of a program module, we now take a closer look at its parts.

Import Lists

The import lists at the beginning of a program module specify entities to be imported from library modules. For example, the import list

```
FROM IO IMPORT RdCard, WrCard, WrStr, WrLn;
```

in **ReverseNumber** indicates that the module needs the entities **RdCard**, **WrCard**, **WrStr**, and **WrLn** exported by the **IO** module.

When a Modula-2 compiler encounters an import list, it must fetch descriptions of the imported entities so that it can perform type-checking whenever the module references one of these entities. The TopSpeed Modula-2 system does this by compiling the definition part of the imported module; in general, TopSpeed Modula-2 compiles the definition part of a library module every time the module is referenced in an import list.

The "**FROM Identifier**" part of an import list may be omitted, in which case the identifiers following the word **IMPORT** must be module names; the effect is to import *all* identifiers that these modules export. For example, **ReverseNumber** could use the following import list:

```
IMPORT IO;
```

This form of import makes available all the entities that **IO** exports.

When a module is imported in this fashion, all references to entities that the module exports must be *qualified* (preceded by the name of the module). For example, if

`ReverseNumber` were to import `IO` without the `FROM` clause, it would appear as follows:

```
MODULE ReverseNumber;
  (* writes the digits of a nonnegative integer in reverse order *)

  IMPORT IO;

  VAR number: CARDINAL;

BEGIN
  IO.WrStr("Enter number to be reversed: ");
  number := IO.RdCard();
  IO.WrLn;

  IO.WrStr("The reversal is: ");
  REPEAT
    IO.WrCard(number MOD 10, 1);
    number := number DIV 10
  UNTIL number = 0;
  IO.WrLn
END ReverseNumber.
```

A name such as `IO.WrStr` is a *qualified identifier*. In theory, qualified identifiers can consist of many identifiers strung together, but in practice a qualified identifier is almost always just two identifiers separated by a period. The name preceding the period (the module name) is a *qualifying identifier*.

Using qualified names for imported entities has advantages and disadvantages. The chief advantage is that the resulting program is easier to modify (the source of each imported entity is immediately apparent). The primary disadvantage is the awkwardness of qualified names. (Appropriate module names can help make qualification less awkward, however, hence the short names—`IO` and `FIO`—for the TopSpeed I/O modules).

Qualified names are mandatory when importing entities with the same name from two library modules. This situation arises often in TopSpeed Modula-2, because many of the procedures exported by the `IO` and `FIO` modules have the same name. For example, both `IO` and `FIO` export procedures named `WrStr` and `WrLn`. It is illegal to export entities with the same name from two different modules, so we cannot use the following import lists:

```
FROM IO IMPORT WrStr, WrLn;
FROM FIO IMPORT Open, Close, WrStr, WrLn; (* illegal *)
```

Importing both modules without the `FROM` clause solves the problem:

```
IMPORT IO;
IMPORT FIO;
```

The program will now contain references to `IO.WrStr`, `IO.WrLn`, `FIO.Open`, `FIO.Close`, `FIO.WrStr`, and `FIO.WrLn`. Notice that `Open` and `Close` must be qualified, even though these names are unambiguous. To avoid this unnecessary qualification, we have the option of importing those entities separately:


```
IMPORT IO;  
IMPORT FIO;  
FROM FIO IMPORT Open, Close;
```

(The order of the import lists is not significant.) Now we can use the names `Open` and `Close` without qualification. This collection of import lists may look illegal (because `Open` and `Close` are accessible both with and without qualification), but it is not.

Blocks

The heart of a program module is a *block*: a collection of declarations followed by a sequence of statements, with the latter enclosed between `BEGIN` and `END`.

A block begins with an optional sequence of *declarations*. Declarations are mandatory in Modula-2: the programmer must declare all names used in a program (other than standard identifiers or imported names).

Modula-2 allows declarations of

- constants
- types
- variables
- procedures (including functions)
- modules

We refer to these as *entities*. Although the five kinds of entities are very different, all have certain properties in common: all are represented by programmer-created identifiers, all must be declared, and all are subject to the scope rules of Modula-2. (As a language extension, TopSpeed Modula-2 allows the declaration of labels; see the *User's Manual* for details.)

Declarations need not appear in any fixed order; for example, Modula-2 allows the programmer to declare several constants, then a module, then a type, then several variables, then more constants. However, a programmer cannot use one entity to declare another without first declaring the former. For example, before a constant can appear in a type declaration, the programmer must declare the constant. (This rule has one exception: The declaration of a pointer type may refer to a type not yet declared—see Chapter 8.)

Statements

After the declarations comes the body of the block: a statement sequence, preceded by the word `BEGIN` and followed by the word `END`. A *statement sequence* consists of one or more statements, separated by semicolons.

Modula-2 provides eleven kinds of statements:

- assignment
- procedure call
- IF
- CASE
- WHILE
- REPEAT
- LOOP
- FOR
- WITH
- EXIT
- RETURN

A statement may also be *empty*.

Statement Sequences and the Empty Statement

Blocks are just one of many places in Modula-2 where statement sequences appear. A general rule is that a statement sequence may appear wherever Modula-2 allows a single statement. This fact, together with the existence of the empty statement, has important consequences for Modula-2 programmers.

The *empty statement* contains no symbols and has no effect. Here's an example:

```
i := RdCard(); ; j := 3 * i + 4;
```

In this sequence of three statements, the middle statement is empty.

The empty statement has two primary uses. First, it allows a statement sequence to be empty (a statement sequence must always contain at least one statement, but that one statement can be empty). Consider the following **WHILE** statement:

```
WHILE RdChar() # ' ' DO  
END;
```

The only purpose of the **WHILE** statement is to call the function **RdChar** repeatedly until it returns the blank character; consequently, the body of the **WHILE** statement is empty.

Second, the empty statement makes semicolon placement less critical. In fact, programmers can insert extra semicolons before or after any statement. Putting an extra semicolon before a statement or after a statement creates a statement sequence; however, a statement sequence may appear wherever Modula-2 allows a single statement, so no problems arise.

There is no harm in using the semicolon to terminate statements rather than separate them. For example, suppose that we rewrite the preceding **WHILE** statement to skip blanks rather than nonblanks:

```
REPEAT
  ch := RdChar()
UNTIL ch # ' ';
```

The statement `ch := RdChar()` forms a statement sequence of length 1. The following is also legal:

```
REPEAT
  ch := RdChar();    (* unnecessary semicolon *)
UNTIL ch # ' ';
```

The body of the **REPEAT** statement now consists of two statements, but the second is empty. Some programmers deliberately use the semicolon as a statement terminator, because this style makes programs easier to modify. In the example, we can insert a new statement after `ch := RdChar()`; without making any other changes; if we had not already ended the statement with a semicolon, we would have had to add one.

Chapter 3

Basic Types

Modula-2 is a *strongly typed* language. Every variable in a Modula-2 program must have a *type* to indicate the range of values that the variable can assume; the compiler checks this type wherever an operation on the variable appears in the program.

Strong typing has several advantages. First, many programming errors result in type mismatches; with strong typing, it is possible to detect these errors during compilation, when they are easiest to fix. Second, types provide information that a Modula-2 compiler needs. In particular, the compiler can—from the type of a variable—deduce the amount of storage the variable requires. Third, the type of a variable serves as an important source of program documentation: by looking at the declaration of a variable, someone reading a program can quickly determine the range of values that the variable can assume.

Some types are *standard*; Modula-2 compilers recognize these types and hence the programmer need not declare them. Of Modula-2's standard types, the most important are the *basic* types, which consist of the numeric types (**INTEGER**, **CARDINAL**, **REAL**, **LONGINT**, **LONGREAL** and—in TopSpeed Modula-2—**SHORTINT**, **SHORTCARD**, and **LONGCARD**), the character type (**CHAR**), and the Boolean type (**BOOLEAN**). This chapter discusses the basic types, introduces the general rules for forming expressions in Modula-2, and shows how to declare constants and variables.

Numeric Types

Five numeric types are standard in Modula-2: **INTEGER**, **CARDINAL**, **REAL**, **LONGINT**, and **LONGREAL**. The **INTEGER** type consists of whole numbers, both positive and negative, in a system-defined range. The **CARDINAL** type is similar, but includes only 0 and a range of positive integers. **REAL** values represent real numbers, both

positive and negative. **LONGINT** provides a larger range of integers than **INTEGER**. **LONGREAL** values are real numbers of greater precision (and possibly greater magnitude) than **REAL** values. TopSpeed Modula-2 also supplies **SHORTINT**, **SHORTCARD**, and **LONGCARD** as language extensions.

The **INTEGER** Type

The range of **INTEGER** values varies from one Modula-2 system to another. The standard functions **MIN** and **MAX**, when applied to the type **INTEGER**, yield the smallest and largest **INTEGER** values. In TopSpeed Modula-2, **MAX (INTEGER)** is 32,767 and **MIN (INTEGER)** is -32,768.

Modula-2 supplies five binary operations for values of type **INTEGER**:

+	addition
-	subtraction
*	multiplication
DIV	quotient
MOD	remainder

The only operations that deserve special mention are **DIV** ("divide") and **MOD** ("modulus"). If *i* and *j* have type **INTEGER**, then *i* **DIV** *j* is the quotient obtained when *i* is divided by *j* (with the fractional portion dropped if the result is not an integer), and *i* **MOD** *j* is the remainder.

Expression	Value
6 DIV 2	3
6 MOD 2	0
1 DIV 3	0
1 MOD 3	1
-3 DIV 2	-1
3 DIV -2	-1

The value of *i* **MOD** *j* is defined only if *i* is nonnegative and *j* is positive.

The operations *****, **DIV**, and **MOD** take precedence over the operations **+** and **-**. Modula-2 performs operations at the same level of precedence from left to right:

Expression	Value
1 - 6 - 4	-9
5 + 8 * 3	29
4 * 10 + 6 DIV 2	43

The programmer can use parentheses to change the order of evaluation:

Expression	Value
1 - (6 - 4)	-1
(5 + 8) * 3	39
4 * (10 + 6) DIV 2	32

The symbols + and - represent unary operators as well as binary operators. However, Modula-2 forbids adjacent operators, so + and - can appear only at the beginning of an arithmetic expression or when preceded by a left parenthesis.

Expression	Value
-9 + 4 DIV 2	-7
12 + -3	not legal (+ and - are adjacent)
12 + (-3)	9

The standard functions **ABS** and **ODD** can be applied to integers. **ABS (i)** is the absolute value of **i**. The value of **ODD (i)** is **TRUE** if **i** is odd, **FALSE** if **i** is even:

Expression	Value
ABS (4)	4
ABS (-4)	4
ODD (2)	FALSE
ODD (-7)	TRUE
ODD (0)	FALSE

Oddly enough, Modula-2 does not provide an **EVEN** function. To test whether **i** is even, use the expression **NOT ODD (i)**.

The **CARDINAL** Type

CARDINAL is similar to **INTEGER**, except that **CARDINAL** includes only nonnegative integers. Most Modula-2 systems store values of type **CARDINAL** in the same amount of space as values of type **INTEGER**. The bit that normally indicates the sign of an **INTEGER** value can store part of the magnitude of a **CARDINAL** number; consequently, the largest **CARDINAL** is normally twice as large as the largest **INTEGER**, plus 1. The expression **MAX (CARDINAL)** gives the largest **CARDINAL** number; the smallest **CARDINAL** is always 0.

In TopSpeed Modula-2, **INTEGER** values occupy 16 bits. **MAX (INTEGER)** is 32,767 ($2^{15} - 1$), because the magnitude of an **INTEGER** value occupies 15 bits; the remaining bit indicates the number's sign. **MAX (CARDINAL)**, however, is 65,535 ($2^{16} - 1$), because the system can use all 16 bits to store the magnitude of a **CARDINAL**.

All **INTEGER** operators are defined for **CARDINAL** operands as well (with the exception of the unary - operation), as is the standard function **ODD**. (**ABS**, however, is not very useful, because the absolute value of a **CARDINAL** number is the number itself.)

We gain several advantages by declaring a variable **i** to have type **CARDINAL** instead of type **INTEGER** if we know that **i** never takes on negative values. First, **i** can take on larger values without overflow. Second, arithmetic operations involving **i** may execute faster on some machines. Third, by declaring **i** to be of type **CARDINAL**, the programmer documents the fact that **i** should never be negative. Best of all, the value of **i** can be checked at run-time if requested by compiler directive. When this option is selected, the program terminates if **i** ever takes on a negative value, serving as an early signal that the program is not behaving as expected.

The **CARDINAL** type also has disadvantages. **CARDINAL** values cannot be mixed freely with **INTEGER** values in expressions, as we will see later in this section; combining **CARDINAL** and **INTEGER** values requires type conversion. Furthermore, the **CARDINAL** type requires careful handling; the programmer must ensure that **CARDINAL** expressions can never be negative. For example, suppose that **i**, **j**, and **k** are variables of type **CARDINAL**; one might expect the **CARDINAL** expression

$$i - j + k$$

to be legal provided that its value is not negative. Unfortunately, the expression is illegal if **i** is less than **j**, because the result of subtracting **j** from **i** is negative (and hence not a **CARDINAL** value). Another problem can occur when the control variable in a **FOR** statement is of type **CARDINAL** (see Chapter 4).

Because of these disadvantages, the **CARDINAL** type is not as useful as might appear at first. Still, it is important in systems programming, because here unsigned numbers are common and the arithmetic performed on them is simple (adding or subtracting 1, for example).

The **REAL** Type

Values of type **REAL** represent real (floating-point) numbers. In TopSpeed Modula-2, **REAL** numbers have magnitudes ranging from $1.2\text{E}-38$ to $3.4\text{E}38$ and a precision of six digits.

The following binary operations may be performed on **REAL** numbers:

+	addition
-	subtraction
*	multiplication
/	division

The **REAL** operators are similar to the **INTEGER** and **CARDINAL** operators, except that **/** replaces **DIV**, and **MOD** is absent. The different division symbol reminds us that dividing two **REAL** values does not produce a remainder; the quotient is as exact as the underlying hardware allows.

Modula-2 allows the use of unary + and - and the **ABS** function (but not **ODD**) with **REAL** numbers.

"Long" and "Short" Numeric Types

Two other numeric types are standard in Modula-2: **LONGINT** and **LONGREAL**. **LONGINT** includes a larger range of integers than **INTEGER**, while **LONGREAL** provides real numbers of greater precision (and usually greater magnitude) than **REAL**. In TopSpeed Modula-2, **LONGINT** values are 32 bits long, so **MAX(LONGINT)** is 2,147,483,647 and **MIN(LONGINT)** is -2,147,483,648. In TopSpeed Modula-2, the magnitude of a **LONGREAL** number can range from $2.3\text{E}-308$ to $1.7\text{E}308$, with 15 digits of precision.

TopSpeed Modula-2 also provides the types **SHORTINT** (8 bits), **SHORTCARD** (8 bits), and **LONGCARD** (32 bits) as extensions to Modula-2. **SHORTINT** values range from -128 to 127. **SHORTCARD** values range from 0 to 255. **LONGCARD** values range from 0 to 4,294,967,295.

Mixed Expressions

Modula-2 does not allow *mixed expressions*; values combined in an expression must have the same type. For example, the addition operator may not have one operand of type **INTEGER** and the other of type **CARDINAL** or **REAL**. (More precisely, operands in an expression must be *compatible*. The slightly weaker requirement of compatibility treats values of a subrange type as though they belonged to the base type. We discuss compatibility in Chapter 6.)

When values of different numeric types are mixed in an expression, the programmer must use conversion functions to convert the operands to the same type. There are several standard functions for type conversion; these include **FLOAT**, **TRUNC**, **ORD**, and **VAL**.

The standard function **FLOAT** converts a **CARDINAL** number to the corresponding **REAL** value. (**FLOAT** can also convert a nonnegative **INTEGER** value to **REAL**.) **TRUNC** converts a **REAL** value to a **CARDINAL** value by dropping the **REAL** number's fractional part:

Expression	Value
TRUNC (6.05)	6
TRUNC (7.5)	7
TRUNC (8.9)	8

To round a **REAL** number x to the closest **CARDINAL** value, use the expression **TRUNC** ($x + 0.5$). (Modula-2 provides no rounding function.)

ORD can be used to convert an **INTEGER** value to type **CARDINAL**. (ORD can also be used to convert certain other types to **CARDINAL**, as we will see later.) We run some risk in converting from **CARDINAL** to **INTEGER**, or vice-versa. An attempt to convert a negative **INTEGER** to **CARDINAL** is illegal, as is an attempt to convert a **CARDINAL** value larger than **MAX (INTEGER)** to **INTEGER**.

The call **VAL (T, X)** returns the value of **X** converted to type **T**, where **T** is a type name. In many Modula-2 systems, **T** is limited to certain types (including **INTEGER** and **CARDINAL**). However, in TopSpeed Modula-2, **VAL** is a general-purpose conversion function; it can convert from any numeric type to any other numeric type. For example, if **i** has type **CARDINAL** and **r** has type **REAL**, then **VAL (REAL, i)** is the same as **FLOAT (i)**, and **VAL (CARDINAL, r)** is the same as **TRUNC (r)**. In particular, **VAL** can be used to convert among "normal" types, "short" types, and "long" types.

Although explicit type conversions may seem to be a nuisance, programmers do not need them very often. Moreover, forcing the programmer to perform the conversion explicitly has certain advantages. The programmer thus becomes aware of type conversions; in other languages, many conversions occur without the programmer's knowledge, which leads to errors. Also, Modula-2 compilers can be smaller and faster if the programmer handles type conversions.

The TopSpeed MATHLIB Module

We saw earlier in this section that Modula-2 provides the **ABS** (absolute value) function. No other mathematical functions are standard in Modula-2. Programs that need additional mathematical functions must import them from a library module. In TopSpeed Modula-2, the **MATHLIB** module provides the following mathematical functions:

Sin (A)	sine of A
Cos (A)	cosine of A
Tan (A)	tangent of A
ASin (A)	arcsine of A
ACos (A)	arccosine of A
ATan (A)	arctangent of A
ATan2 (X, Y)	arctangent of X/Y
SinH (A)	hyperbolic sine of A
CosH (A)	hyperbolic cosine of A
TanH (A)	hyperbolic tangent of A
Log (A)	logarithm (base <i>e</i>) of A
Log10 (A)	logarithm (base 10) of A
Pow (X, Y)	X raised to the power Y
Exp (A)	<i>e</i> raised to the power A
Mod (X, Y)	X mod Y
Sqrt (A)	square root of A

A, **X**, and **Y** must be of type **LONGREAL**; all functions return **LONGREAL** results. The **Sin**, **Cos**, and **Tan** functions expect values in radians, not degrees.

Example Program: Converting a Fahrenheit Temperature to Celsius

The following program, **ConvertToCelsius**, converts a Fahrenheit temperature to Celsius. Because we normally express temperatures as integers, the program reads the Fahrenheit temperature as an integer and prints the Celsius equivalent as an integer. Inside **ConvertToCelsius**, however, we use real arithmetic so that the result can be rounded to the nearest integer. Therefore, **ConvertToCelsius** uses the **VAL** function to convert the input value to type **REAL** and the output value to type **INTEGER**.

```
MODULE ConvertToCelsius;
  (* converts a Fahrenheit temperature to Celsius *)

  FROM IO IMPORT RdInt, WrInt, WrStr, WrLn;

  VAR fahrenheit: INTEGER;
      celsius: REAL;

BEGIN
  WrStr("Enter Fahrenheit temperature: ");
  fahrenheit := RdInt();

  celsius := (VAL(REAL, fahrenheit) - 32.0) * 5.0 / 9.0;

  WrStr("Celsius equivalent is: ");
  WrInt(VAL(INTEGER, celsius + 0.5), 1);      (* round answer *)
  WrLn
END ConvertToCelsius.
```

The CHAR Type

Values of type **CHAR** include letters, digits, punctuation marks, and nonprinting (control) characters. **CHAR** is not the same for all Modula-2 systems, because its values are the characters provided by the underlying hardware. TopSpeed Modula-2 uses the ASCII character set.

In Modula-2, strings of length 1 represent **CHAR** values. Thus, the programmer can enclose a character between either single quotes ('**A**', for example) or double quotes ("**A**"). The string **' '** represents the character **'** and the string **' "'** represents the character **"**.

The values of type **CHAR** are ordered, but the ordering depends on the character set in use. The position of a character in this ordering is said to be its *ordinal number*. In ASCII, the first character is *nul*, which has ordinal number 0, and the last character is *del*, which has ordinal number 127.

Representing Nonprinting Characters

The ASCII characters with ordinal numbers between 0 and 31 cannot be printed (nor can the character with ordinal number 127). The nonprinting characters include the control characters, ^A through ^Z, which we can generate at the keyboard by holding down the Control key and striking one of the letters between A and Z. Programmers cannot include nonprinting characters in programs using the normal single- or double-quote convention. In order to allow nonprinting characters to appear in programs, Modula-2 allows an alternate representation of a character: an octal number (the ordinal number of the character), followed by the letter C. For example, the following statement writes the line-feed character:

```
WrChar(12C);    (* write line-feed character *)
```

Here are the Modula-2 representations of some common nonprinting ASCII characters:

<i>Modula-2 Symbol</i>	<i>ASCII Character</i>
0C	null
7C	bell (^G)
10C	backspace (^H)
11C	horizontal tab (^I)
12C	line feed (^J)
15C	carriage return (^M)
33C	escape
177C	delete

The ORD and CHR Functions

The standard function **ORD** computes the ordinal number of a character. For example, if the character set is ASCII, then **ORD('A')** is 65 and **ORD('a')** is 97. The standard function **CHR** yields the inverse of **ORD**. Thus, **CHR(65)** is 'A' and **CHR(97)** is 'a'.

Modula-2 provides no operations for **CHAR** values other than the relational operations =, # (or <>), <, <=, >, and >=. In particular, the arithmetic operators do not allow character operands. However, the programmer can use **ORD** and **CHR** to perform arithmetic on characters. For example, to compute the offset between the uppercase letters and the lowercase letters in ASCII, we could *not* use the expression

```
'a' - 'A'
```

That expression is illegal; instead, we would use

```
ORD('a') - ORD('A')
```


The CAP Function

Modula-2 provides a standard function called **CAP** that is often useful when dealing with character data. When given a lowercase letter, **CAP** returns the corresponding uppercase letter; when given any other character, **CAP** returns the same character unchanged. Thus, **CAP ('a')** is **'A'** and **CAP ('A')** is **'A'**.

Example Program: Encoding a Message

The program **Encode** encodes a message as it is typed at the keyboard. **Encode** uses a simple "rotation" scheme to perform the encoding: it replaces the letters from A through M by N through Z, respectively, and replaces the letters from N through Z by A through M. Characters that are not letters do not change. The program has two key statements. One converts an uppercase letter to its encoded version; the other encodes a lowercase letter. The following statement does the uppercase encoding:

```
ch := CHR((ORD(ch) - ORD('A') + 13) MOD 26 + ORD('A'));
```

The idea is to perform a "circular shift" of the letter **ch** by adding 13 to **ORD(ch)**, taking the result modulo 26, then converting back to character form by using **CHR**. This operation will work only if the program first subtracts **ORD('A')** from **ORD(ch)**—to scale it into the range between 0 and 25—and adds it after performing the **MOD** operation. (This strategy assumes that the uppercase letters occupy contiguous positions in the underlying character set—as is true in ASCII, but not in every character set.)

Encode reads characters typed at the keyboard until the user presses the escape key (which generates the character 33C). The program uses the **RdKey** and **WrChar** procedures (imported from **IO**) to read and write one character at a time. **RdKey** is similar to **RdChar**, but does not echo characters typed at the keyboard; consequently, the user sees only the encoded version of the message, not the original version.

```
MODULE Encode;
  (* encodes a message; assumes ASCII character set *)

  FROM IO IMPORT RdKey, WrChar, WrStr, WrLn;

  CONST esc = 33C;  (* escape key *)

  VAR ch: CHAR;

  BEGIN
    WrStr("Enter message to be encoded.");
    WrLn;
    WrStr("Indicate end of message by pressing Esc.");
    WrLn;
```



```

ch := RdKey();
WHILE ch # esc DO
  IF ('A' <= ch) & (ch <= 'Z') THEN
    ch := CHR((ORD(ch) - ORD('A') + 13) MOD 26 + ORD('A'))
  ELSIF ('a' <= ch) & (ch <= 'z') THEN
    ch := CHR((ORD(ch) - ORD('a') + 13) MOD 26 + ORD('a'))
  ENDIF;
  WrChar(ch);
  ch := RdKey();
END
END Encode.

```

One property of this code is that a second application of the encoding algorithm undoes the result of the first application. Consequently, we can also use **Encode** for decoding.

The BOOLEAN Type

The **BOOLEAN** type has only two values, denoted by the standard identifiers **TRUE** and **FALSE**. The *relational operators* (equal, not equal, less than, and so on) produce **BOOLEAN** values. The three *logical operators* (**AND**, **OR**, and **NOT**) require **BOOLEAN** operands and produce a **BOOLEAN** result.

Relational Operators

Modula-2 provides seven relational operators:

=	equal
<> or #	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
IN	set membership

The symbols **<>** and **#** both represent the "not equal" operator and are interchangeable. (The latter symbol is meant to suggest the $\neq$ relation of mathematics.)

We can use the relational operators (other than **IN**) to compare values of any basic type, enumeration type, or subrange type.

Modula-2 considers the **BOOLEAN** value **FALSE** to be less than the value **TRUE**. **CHAR** values are compared using their ordinal numbers:

Expression	Value
'a' < 'b'	TRUE (ORD('a') is 97 and ORD('b') is 98)
'a' < 'A'	FALSE (ORD('a') is 97 and ORD('A') is 65)

In order to compare two values using one of the operators =, <>, <, <=, >, or >=, the values must have compatible types. In essence, the operands must have the same type—the only exception is for subrange types.

We can also use the operators = and <> to compare sets and pointers. (As a language extension, TopSpeed Modula-2 allows the use of = and <> with values of *any* type, including arrays and records.) The relational operator **IN** is meaningful only in conjunction with sets. The operators <= and >= have a special meaning (set inclusion) when we use them to compare sets.

Logical Operators

Modula-2 provides three logical operators: **AND**, **OR**, and **NOT**. These operators require **BOOLEAN** operands and produce a **BOOLEAN** result. The symbol & is a synonym for **AND**; we can use the two interchangeably. Similarly, ~ is a synonym for **NOT**.

The following rules define the values of the expressions p **AND** q , p **OR** q , and **NOT** p :

<i>Expression</i>	<i>Has the Value TRUE If</i>
p AND q	both p and q are TRUE
p OR q	p or q (or both) is TRUE
NOT p	p is FALSE

The expressions are **FALSE** in all other cases.

In expressions involving two or more logical operators, **NOT** takes precedence over **AND**, and **AND** takes precedence over **OR**. Both **AND** and **OR** are left-associative. Thus, the expression

p **OR** q **AND** **NOT** r **OR** s

is equivalent to

$(p$ **OR** $(q$ **AND** $($ **NOT** $r)))$ **OR** s

Modula-2 requires "short-circuit" evaluation of the **AND** and **OR** operators. In the expression p **AND** q , the value of p is first checked; if it is **FALSE**, then the expression is automatically **FALSE**, so q is not evaluated. Similarly, p **OR** q has the value **TRUE** if p is **TRUE**; q is not evaluated. For example, consider the following expression:

$(i < j)$ **AND** $(j < k)$ **OR** $(k < j)$ **AND** $(j < i)$

If $(i < j)$ is **FALSE**, then $(j < k)$ is not evaluated; if $(k < j)$ is also **FALSE**, then $(j < i)$ is not evaluated either. However, if $(i < j)$ and $(j < k)$ are both **TRUE**, then neither $(k < j)$ nor $(j < i)$ are evaluated.

Short-circuit evaluation of **BOOLEAN** expressions can be very useful. For example, in the expression

```
(x > 0.0) AND (y / x > 0.0)
```

the value of y / x will not be computed if x is less than or equal to zero, thus preventing a division by zero.

Expressions

Modula-2's arithmetic, relational, and logical operators allow the construction of complicated expressions. To ensure that each expression has a unique meaning, Modula-2 provides rules for operator precedence and associativity. In Modula-2, operators are divided into four levels of precedence. The **NOT** operator has the highest precedence, followed by the "multiplication" operators (which include the logical operator **AND**), the "addition" operators (which include the logical operator **OR**), and finally the relational operators:

highest:	NOT ~
	* / DIV MOD AND &
	+ - OR
lowest:	= # <> < <= > >= IN

All binary operators are left-associative. Thus, operations at the same level of precedence are performed from left to right.

The rules for precedence and associativity come into play whenever an expression contains more than one operator. If the expression contains operators at different levels of precedence, the program performs the operations in the order just described. If several operations are at the same precedence level, the associativity rule dictates performing them from left to right. For example,

<i>Expression</i>	<i>Order of Evaluation</i>
$i * 3 - j \text{ DIV } 2$	$(i * 3) - (j \text{ DIV } 2)$
$i \text{ DIV } j * k$	$(i \text{ DIV } j) * k$
$-i - j - k$	$((-i) - j) - k$
$p \text{ OR } q \text{ AND } r$	$p \text{ OR } (q \text{ AND } r)$
$p \text{ AND } q \text{ AND } r$	$(p \text{ AND } q) \text{ AND } r$
$p \text{ AND NOT } q \text{ OR } r$	$(p \text{ AND } (\text{NOT } q)) \text{ OR } r$
$i + 2 * j < k + 1$	$(i + (2 * j)) < (k + 1)$

Note, however, that in an expression such as

```
p AND NOT q
```


the expression **p** is evaluated before **NOT q**, even though **NOT** takes precedence over **AND**. Because Modula-2 requires "short-circuit" evaluation of **BOOLEAN** expressions, **NOT q** is not evaluated at all if **p** is **FALSE**.

We can use parentheses to change the order of evaluation within an expression. When in doubt, and particularly when expressions are complicated, use parentheses liberally.

The fact that the relational operators have lower precedence than the logical operators means that relational expressions must be enclosed in parentheses when combined using logical operators. For example, if **i**, **j**, and **k** are **INTEGER** variables, the expression

```
i < j AND j < k
```

is not legal in Modula-2; **AND** has higher precedence than **<**, so a Modula-2 compiler interprets the expression as **(i < (j AND j)) < k**. When parentheses enclose the relational expressions, the problem vanishes:

```
(i < j) AND (j < k)
```

Constant Expressions

In Modula-2, a *constant expression* may appear wherever the language allows a constant. A constant expression has the same syntax as an expression. Unlike an ordinary expression, however, a constant expression must contain only constant operands, never variables. Furthermore, a compiler must be able to evaluate a constant expression without executing the program. Constant expressions can include calls of standard functions such as **MAX** and **MIN**, but may not contain calls of user-defined functions. The following are examples of constant expressions:

```
13.5 * 1.0567 / 4.0  
(MAX(CARDINAL) - 1) DIV 2  
('+' < '-' ) OR ('Z' >= 'z')
```

The only identifiers that may appear in constant expressions (other than the names of standard functions) are identifiers bound to constant values by means of constant declarations.

Constant and Variable Declarations

Modula-2 provides two ways to associate an identifier with a data value: by a *constant declaration* or by a *variable declaration*. A constant declaration specifies an identifier and a constant value; the association between the identifier and the value is permanent and cannot be changed by assignment. A variable declaration specifies an identifier and a type. No fixed value is associated with the identifier; its value changes during program execution as the result of assignment.

Constant Declarations

A constant declaration binds an identifier to a constant value; each subsequent appearance of the identifier is equivalent to an appearance of the constant value that it represents. Here are several constant declarations:

```
CONST speedOfLight = 1.86E5;    (* in miles per second *)
      absoluteZero = -273.1;    (* Celsius *)
      monthlySalesGoal = 1000;
      questionMark = '?';
      lineFeed = 12C;
      debugOption = TRUE;
      errorMessage = "END expected";
```

The reserved word **CONST** must precede a sequence of constant declarations; a semicolon must terminate each declaration.

As the example shows, constants can represent numbers, strings, and values of types **CHAR** and **BOOLEAN**. Constants can also represent enumeration values and sets. As a language extension, TopSpeed Modula-2 allows the definition of array and record constants (called *aggregates*); see Chapter 7 for details.

Arbitrary constant expressions may appear to the right of the equals sign:

```
CONST pi = 3.14159;
      radius = 10.0;
      area = pi * radius * radius;
      first = 0;
      last = 99;
      length = last - first + 1;
```

The programmer must declare a constant before using it to declare other constants. For example, the declaration of **area** relies on the previously declared constants **pi** and **radius**. An attempt to declare **area** before **pi** and **radius** would be illegal:

```
CONST area = pi * radius * radius;  (* illegal *)
      pi = 3.14159;
      radius = 10.0;
```

Giving a name to a constant value has several advantages:

- The name helps document the meaning of the constant value. (**speedOfLight** is more helpful to someone reading a program than **1.86E5** would be.)
- To change the value of the constant, the programmer must change only the declaration of the constant, not every occurrence of the constant in the program. (**speedOfLight** is unlikely to change, but **monthlySalesGoal** probably will.)
- When the same constant value occurs more than once in a program, some of its appearances may be incorrect because of typographical errors. (Long numerical values such as **2.7182818284** are prone to this sort of error.) If an identifier replaces each occurrence of the value, the risk of error greatly diminishes, because the value

itself appears only once in the program (in the constant declaration that binds the identifier to the value).

As a language extension, TopSpeed Modula-2 provides *alias* declarations, which resemble constant declarations. An alias declaration allows us to give a new name to a previously defined entity. See the *User's Manual* for more information.

Variable Declarations

Modula-2 requires the programmer to declare every variable. A variable declaration consists of a list of identifiers and a type; the variables are permanently bound to that type and can be assigned only values belonging to the type. The reserved word **VAR** must precede a sequence of variable declarations, and a semicolon must terminate each declaration. For example, here is a group of three variable declarations:

```
VAR i1, i2, i3: INTEGER;
    epsilon: REAL;
    oldChar, newChar: CHAR;
```

Example Program: Computing the Circumference and Area of a Circle

The following program computes the circumference and area of a circle. We declare the number 3.14159 as a constant, so it needs to appear only once in the program.

```
MODULE Circle;
  (* computes the circumference and area of a circle *)

  FROM IO IMPORT WrStr, WrLn, RdReal, WrReal;

  CONST pi = 3.14159;
        twoTimesPi = 2.0 * pi;

  VAR radius: REAL;

BEGIN
  WrStr("Enter radius: ");
  radius := RdReal();

  WrStr("Circumference is: ");
  WrReal(twoTimesPi * radius, 6, 1);
  WrLn;

  WrStr("Area is: ");
  WrReal(pi * radius * radius, 6, 1);
  WrLn
END Circle.
```

WrReal writes a **REAL** number in scientific notation. The number 6 indicates that six digits of precision are desired; the number 1 indicates that the output is to be written in a field of minimum size.

Chapter 4

Statements

This chapter presents eight of Modula-2's eleven statements. The first section covers the assignment statement. We devote the remainder of the chapter to *control structures*, which control the order in which statements are executed. We can divide Modula-2's control structures into three groups. Statements in the first group (the **IF** and **CASE** statements) are for *selection*: choosing between two or more computation paths. The other control structures provide ways to perform *iteration* or *looping*—the repeated execution of a sequence of statements. Some of these statements (**WHILE**, **REPEAT**, and **LOOP**) allow *indefinite iteration*, in which the loop executes an indefinite number of times (typically under the control of a **BOOLEAN** expression). One control structure (the **FOR** statement) provides *definite iteration*, in which we know the number of iterations before the loop executes.

All of these statements are *structured*: they contain other statements as components. (Modula-2 has one other structured statement, **WITH**, but it is not a control structure.) The structured statements all end with the reserved word **END**. For example, the **WHILE** statement has the form

```
WHILE Expression DO
    StatementSequence
END
```

When we write a structured statement, we always put each statement inside, as well as the terminating **END**, on a separate line. To make a structured statement easier to read, we align its final **END** with the first line of the statement and indent the statements inside two spaces. (Some programmers prefer to indent three or more spaces; some also indent the **END** the same number of spaces as the inner statements.)

In many programming languages, some forms of selection and iteration require **GOTO** statements. Modula-2's wide variety of control structures renders unnecessary this use of the **GOTO**. (In theory, **IF** and **WHILE** statements alone are sufficient to write any program without using a **GOTO**; in practice, other control structures are necessary for program clarity.) Other common uses of the **GOTO** are to break out of a loop, return from a procedure prematurely, or terminate a program when an error is detected. Modula-2 provides three constructs to handle these situations:

- the **EXIT** statement
- the **RETURN** statement
- the **HALT** procedure

We can view each of these as a restricted **GOTO** statement.

The **EXIT** statement can appear only inside **LOOP** statements. Executing an **EXIT** statement terminates execution of the **LOOP** statement in which the **EXIT** appears; thus, **EXIT** is effectively a **GOTO** statement whose target is the statement immediately following the **LOOP** statement.

The **RETURN** statement can appear in procedures and modules. Executing a **RETURN** statement terminates execution of the procedure or module in which the statement appears. A **RETURN** statement is effectively a **GOTO** statement whose target is the end of the enclosing procedure or module.

The standard procedure **HALT** causes the entire program to terminate, thus acting like a **GOTO** whose target is the end of the program module.

TopSpeed Modula-2 provides the **GOTO** statement as an extension to Modula-2 (see the *User's Manual* for details). However, with **EXIT**, **RETURN**, and **HALT** available, **GOTO** statements should not be needed except in exceptional circumstances. Because **EXIT**, **RETURN**, and **HALT** transfer control only in the forward direction, they are unlikely to cause the tangled flow of control that can result from the uncontrolled use of **GOTO** statements.

The Assignment Statement

The Modula-2 assignment statement has the form

Designator := Expression

The designator denotes a storage location—a place to store a value. Most of the time, a designator is just the name of a variable. A designator can also be

- an array component
- a record field
- an anonymous variable named by dereferencing a pointer variable

When an assignment statement is executed, the expression to the right of the `:=` symbol is evaluated and the result stored in the location indicated by the designator. The definition of Modula-2 does not specify the relative order in which the designator and the expression are evaluated.

Here are some examples of assignment statements:

```
VAR temperature: INTEGER;
    high, low, average: REAL;
    grade: CHAR;
    done, lessThan: BOOLEAN;
    .
    .
    .
temperature := -10;
low := 1.5;
high := 3.75;
average := (low + high) / 2.0;
grade := 'B';
done := FALSE;
lessThan := low < high;
```

In general, the types of the designator and the expression in an assignment statement must be *assignment-compatible*. Assignment compatibility is somewhat weaker than requiring that the types be the same. In particular, **INTEGER** and **CARDINAL** are assignment-compatible, allowing assignment of an **INTEGER** value to a **CARDINAL** variable and vice-versa. Of course, such an assignment is illegal if the expression is negative and the designator has type **CARDINAL** or if the expression is a **CARDINAL** value greater than **MAX (INTEGER)** and the designator is **INTEGER**. (A Modula-2 compiler cannot usually catch such an error, which must be detected during program execution.) We define assignment compatibility more precisely in Chapter 6.

The rule of assignment compatibility has one exception: Modula-2 allows the assignment of a string literal to any string variable of equal or greater length (see Chapter 7).

The INC and DEC Procedures

Modula-2 furnishes two standard procedures, **INC** and **DEC**, that can often take the place of an assignment statement. When given a variable *i* of type **INTEGER** or **CARDINAL**, **INC** increments the value of *i*, while **DEC** decrements its value:

```
INC(i);      (* equivalent to i := i + 1 *)
DEC(i);      (* equivalent to i := i - 1 *)
```

TopSpeed Modula-2 allows the use of **INC** and **DEC** with a variable of any whole number type, whether it be "normal," "short," or "long."

INC and **DEC** allow an optional second parameter, which specifies the amount by which to increment or decrement the first parameter:

```
INC(i, n);      (* equivalent to i := i + n *)
DEC(i, n);      (* equivalent to i := i - n *)
```

INC and **DEC** can be used with variables of type **CHAR**:

```
INC(ch);        (* equivalent to ch := CHR(ORD(ch) + 1) *)
DEC(ch);        (* equivalent to ch := CHR(ORD(ch) - 1) *)
INC(ch, n);     (* equivalent to ch := CHR(ORD(ch) + n) *)
DEC(ch, n);     (* equivalent to ch := CHR(ORD(ch) - n) *)
```

INC(ch) replaces **ch** by the character next in the ordering of **CHAR** values. **DEC(ch)** replaces **ch** by the character prior to it in the ordering. Thus, if **ch** contains 'b' initially, then **ch** would have the value 'c' after **INC(ch)** and 'a' after **DEC(ch)**. **INC** and **DEC** have a similar effect when used with variables of an enumeration type.

INC and **DEC** are useful for at least three reasons. First, they provide a convenient way to step through values of type **CHAR** or an enumeration type. Second, some Modula-2 compilers may generate more efficient object code for a call of **INC** or **DEC** than for an ordinary assignment statement. Third, a call of **INC** or **DEC** is simpler than the corresponding assignment statement, particularly if the designator on the left side of the assignment is complicated.

The IF and CASE Statements

Modula-2 provides two control structures for selection: the **IF** statement and the **CASE** statement. The **IF** statement chooses among alternatives based on the value of one or more **BOOLEAN** expressions. The **CASE** statement makes its choice based on the value of a single *case expression*.

The IF Statement

In its basic form, the **IF** statement allows a program to choose between two alternatives based on the value of a **BOOLEAN** expression:

```
IF divisor # 0 THEN
    result := dividend DIV divisor
ELSE
    HALT
END;
```

If the condition **divisor # 0** has the value **TRUE**, the statement between **THEN** and **ELSE** will be executed; if the condition is **FALSE**, the statement between **ELSE** and **END** is performed instead.

We can omit the **ELSE** clause if no action is to be taken when the condition is false:

```
IF divisor = 0 THEN
  HALT
END;
result := dividend DIV divisor;
```

The word **END** must terminate every **IF** statement, no matter how simple.

Any number of statements can appear between **THEN** and **ELSE** and between **ELSE** and **END**:

```
IF divisor # 0 THEN
  result := dividend DIV divisor;
  remainder := dividend MOD divisor
ELSE
  WrStr("Error: attempted division by zero");
  Writeln;
  HALT
END;
```

In general, a statement sequence may appear wherever Modula-2 allows a single statement.

IF statements can be nested. For example, the following **IF** statement will find the largest of the three numbers **num1**, **num2**, and **num3**:

```
IF num1 < num2 THEN
  IF num2 < num3 THEN
    largest := num3
  ELSE
    largest := num2
  END
ELSE
  IF num1 < num3 THEN
    largest := num3
  ELSE
    largest := num1
  END
END;
```

Programs often perform a series of tests, only one of which will succeed. For example, suppose we need to determine the sign of the value stored in the variable **number**. We could use nested **IF** statements for this purpose:

```
IF number < 0 THEN
  sign := -1
ELSE
  IF number > 0 THEN
    sign := +1
  ELSE
    sign := 0
  END
END;
```

However, Modula-2 provides a more concise way to express the same series of tests:


```

IF number < 0 THEN
    sign := -1
ELSIF number > 0 THEN
    sign := +1
ELSE
    sign := 0
END;

```

Using **ELSIF** avoids the many **ENDs** that would otherwise appear at the end of a deeply nested **IF** statement. **ELSIF** clauses also indicate that the **IF** statement has a simple structure: it merely evaluates a series of conditions until one of the conditions has the value **TRUE**.

An **IF** statement may have as many **ELSIF** clauses as desired (including none), but at most one **ELSE** clause. The expression following the words **IF** or **ELSIF** must be of type **BOOLEAN**.

The CASE Statement

Consider the problem of translating a letter on a telephone dial or key to the corresponding digit. Assume that the variable **letter** stores a capital letter other than **Q** or **Z** (the letters **Q** and **Z** don't appear on a telephone dial). We must map the letter to one of eight digits, so we could use an **IF** statement controlled by eight **BOOLEAN** expressions:

```

IF (letter = 'A') OR (letter = 'B') OR (letter = 'C') THEN
    digit := 2
ELSIF (letter = 'D') OR (letter = 'E') OR (letter = 'F') THEN
    digit := 3
ELSIF (letter = 'G') OR (letter = 'H') OR (letter = 'I') THEN
    digit := 4
ELSIF (letter = 'J') OR (letter = 'K') OR (letter = 'L') THEN
    digit := 5
ELSIF (letter = 'M') OR (letter = 'N') OR (letter = 'O') THEN
    digit := 6
ELSIF (letter = 'P') OR (letter = 'R') OR (letter = 'S') THEN
    digit := 7
ELSIF (letter = 'T') OR (letter = 'U') OR (letter = 'V') THEN
    digit := 8
ELSIF (letter = 'W') OR (letter = 'X') OR (letter = 'Y') THEN
    digit := 9
END;

```

We can replace this rather lengthy **IF** statement with a much simpler **CASE** statement:

```

CASE letter OF
    'A', 'B', 'C': digit := 2 |
    'D', 'E', 'F': digit := 3 |
    'G', 'H', 'I': digit := 4 |
    'J', 'K', 'L': digit := 5 |
    'M', 'N', 'O': digit := 6 |
    'P', 'R', 'S': digit := 7 |
    'T', 'U', 'V': digit := 8 |
    'W', 'X', 'Y': digit := 9
END;

```


A **CASE** statement contains a *case expression* (between the words **CASE** and **OF**) and a series of *cases* separated by the **|** symbol. Each case consists of a series of values (*case labels*) followed by a statement sequence. When a **CASE** statement is executed, the value of the case expression is computed and compared to the case labels. If the value matches one of the labels, the corresponding statement sequence is executed. For example, if the value of **letter** matches either 'A', 'B', or 'C', the statement **digit := 2** will be executed.

If it is possible that the value of the case expression might not match any of the case labels, a **CASE** statement can include an **ELSE** clause. In our example, if **letter** might have a value other than the ones listed, the **CASE** statement should have an **ELSE** clause:

```
CASE letter OF
  'A', 'B', 'C': digit := 2 |
  'D', 'E', 'F': digit := 3 |
  'G', 'H', 'I': digit := 4 |
  'J', 'K', 'L': digit := 5 |
  'M', 'N', 'O': digit := 6 |
  'P', 'R', 'S': digit := 7 |
  'T', 'U', 'V': digit := 8 |
  'W', 'X', 'Y': digit := 9
ELSE
  WrStr("Error: illegal character in CASE statement");
  WrLn
END;
```

Case labels are most often just constants, but may be constant expressions. (Remember Modula-2's general rule: a constant expression may appear wherever the language allows a constant.) We can abbreviate a sequence of consecutive case labels by writing just the first and last label, separated by two periods:

```
CASE letter OF
  'A'..'C':      digit := 2 |
  'D'..'F':      digit := 3 |
  'G'..'I':      digit := 4 |
  'J'..'L':      digit := 5 |
  'M'..'O':      digit := 6 |
  'P', 'R'..'S': digit := 7 |
  'T'..'V':      digit := 8 |
  'W'..'Y':      digit := 9
END;
```

No label may appear more than once in a **CASE** statement.

The type of the case expression must be a basic type (other than **REAL** or **LONGREAL**), an enumeration type, or a subrange type. In TopSpeed Modula-2, the case expression may not be of type **LONGINT** or **LONGCARD**. The type of each label must be compatible with the type of the case expression; hence, labels must have the same type as the case expression (or the same base type, if the case expression has a subrange type).

A style note: Some programmers prefer to put the `|` symbol to the left of each case (the *User's Manual* follows this convention):

```
CASE score OF
| 90..100: letterGrade := 'A'
| 80..89:  letterGrade := 'B'
| 70..79:  letterGrade := 'C'
| 60..69:  letterGrade := 'D'
| 0..59:   letterGrade := 'F'
END;
```

This convention takes advantage of the fact that a case may be empty. The existence of the empty case makes the placement of `|` symbols less critical (much as the existence of the empty statement makes the placement of semicolons less critical). More than one `|` symbol may separate cases, and extra `|` symbols may precede the first case or follow the last case.

In a large **CASE** statement, putting the `|` symbol at the left makes it easier to find the beginning of each case. It also makes **CASE** statements easier to modify, because the programmer need not move `|` symbols when adding statements to a case or deleting statements from a case.

The WHILE, REPEAT, and LOOP Statements

The **WHILE**, **REPEAT**, and **LOOP** statements are closely related; they all provide a means of indefinite iteration. The first two allow a sequence of statements (the *body* of the loop) to execute repeatedly under the control of a **BOOLEAN** expression, while the **LOOP** statement allows a statement sequence to execute repeatedly until an **EXIT** statement is reached.

The WHILE Statement

The **WHILE** statement allows a sequence of statements to execute repeatedly as long as a stated condition remains true. For example, suppose that we want to read a series of integers and find their sum, stopping when we read a negative value. Using a **WHILE** statement, we can write

```
sum := 0;
i := RdInt();
WHILE i >= 0 DO
    sum := sum + i;
    i := RdInt()
END;
```

When the **WHILE** statement is first reached, the expression `i >= 0` is evaluated. If its value is **TRUE**, then the body of the loop (the statements `sum := sum + i` and `i := RdInt()`) is executed and the expression is evaluated again. If the expression is again

TRUE, then the body is executed once more. This process continues until the value of the expression is **FALSE**. At this point, the loop terminates and the statement following the **WHILE** statement will be executed next.

The most important fact to remember about the **WHILE** statement is that its termination condition is tested *before* the body of the loop is executed. If the condition is false when control reaches the **WHILE** statement, then the body is not executed at all. Note also that, if the condition is true initially, the **WHILE** statement will not terminate unless the statements in the body change the value of the condition.

The REPEAT Statement

The **REPEAT** statement is similar to the **WHILE** statement, except that the body of the loop is executed *until* a stated condition becomes true; furthermore, the condition is tested *after* each execution of the loop body, not before. The body of a **REPEAT** statement is always executed at least once. The syntax of the **REPEAT** statement mirrors the fact that the condition is tested after the body of the loop has been executed:

```
REPEAT  
    StatementSequence  
UNTIL Expression
```

The expression after the word **UNTIL** must be of type **BOOLEAN**.

The **REPEAT** statement is useful when the first execution of the loop body establishes values that the termination condition is to test. For example, consider the problem of prompting the user to enter a value within a particular range (between 1 and 10, say). If the value typed by the user is invalid, the program should repeatedly prompt the user to enter valid input. Using the **WHILE** statement, we might write

```
WrStr("Enter a number between 1 and 10: ");  
number := RdCard();  
WHILE NOT ((1 <= number) AND (number <= 10)) DO  
    WrStr("Enter a number between 1 and 10: ");  
    number := RdCard();  
END;
```

Unfortunately, the two statements in the body of the **WHILE** statement also appear before the **WHILE** statement to give **number** an initial value. We could eliminate them by giving **number** an initial value not between 1 and 10. A better solution is to use a **REPEAT** statement instead of a **WHILE** statement:

```
REPEAT  
    WrStr("Enter a number between 1 and 10: ");  
    number := RdCard();  
UNTIL (1 <= number) AND (number <= 10);
```

The first execution of the loop gives **number** a value so that we need not initialize it above the loop.

The LOOP Statement

WHILE and **REPEAT** statements are natural for expressing most loops. Yet sometimes these statements are awkward to use:

- when the programmer wants an infinite loop
- when the condition for loop exit occurs in the middle of the loop, not at the beginning or end
- when the loop has more than one exit point

The **LOOP** statement handles these situations:

```

LOOP
    StatementSequence
END

```

A **LOOP** statement allows the statements in its body to execute indefinitely. Executing an **EXIT** statement causes the loop to terminate. (The loop can also be terminated in a more drastic fashion by executing a **RETURN** statement or calling the **HALT** procedure.)

Let's take a closer look at the three situations that call for a **LOOP** statement.

1. An Infinite Loop.

Although most programs eventually terminate, a few programs should never stop (for example, an operating system or a program that controls a digital clock). The heart of such a program is an infinite loop, which we express in Modula-2 as a **LOOP** statement whose body contains no **EXIT** statements.

2. An Exit from the Middle of a Loop.

Recall the example that we used earlier to illustrate the **WHILE** statement:

```

sum := 0;
i := RdInt();
WHILE i >= 0 DO
    sum := sum + i;
    i := RdInt()
END;

```

The statement `i := RdInt()` appears twice: once before the loop (to read the first value of `i`) and once inside the loop (to read the next value of `i`). This duplication is typical of **WHILE** statements, because the test for termination must come at the beginning of the loop. By using a **LOOP** statement and moving the test for termination to the middle of the loop, we can eliminate the first call of `RdInt`:


```

sum := 0;
LOOP
  i := RdInt();
  IF i < 0 THEN EXIT END;
  sum := sum + i
END;

```

In this example, we gain little by using a **LOOP** statement instead of a **WHILE** statement. But if we can save several statements in this fashion, the program becomes easier to read and modify.

IF statements often control **EXIT** statements:

```

IF i < 0 THEN EXIT END;

```

When we use an **IF** statement in this fashion, we write the entire statement on a single line instead of three lines; this construct is so common that it is worth making an exception to our usual style.

3. Multiple Exit Points.

As before, suppose that we are reading integers and computing their sum. To this point, we have ignored the possibility that no more integers remain. If so, we want to exit the loop. We might also want to print a message indicating that the loop terminated abnormally. When using the **IO** module, a successful read or write assigns the value **TRUE** to the variable **OK** (which we must import from **IO**). Thus, if **OK** is **FALSE** after a call of **RdInt**, we have read all the integers. Knowing this, we can write a **LOOP** statement with two exits:

```

sum := 0;
LOOP
  i := RdInt();
  IF NOT OK THEN
    WrStr("No negative value was found");
    Writeln;
    EXIT
  END;
  IF i < 0 THEN EXIT END;
  sum := sum + i
END;

```

Several points about the **EXIT** statement are worth mentioning:

- Modula-2 does not limit the number of **EXIT** statements that may appear in a **LOOP** statement.
- When an **EXIT** statement appears in a **LOOP** statement nested within another **LOOP** statement, executing the **EXIT** statement terminates only the inner loop.
- **EXIT** statements can appear only within **LOOP** statements.
- We cannot use an **EXIT** statement to exit from a **WHILE**, **REPEAT**, or **FOR** statement, unless that statement appears within the body of a **LOOP** statement.

Example Program: Testing the MATHLIB Module

When using a library module, programmers sometimes find it helpful to have a program that tests the module's behavior. Such a program certainly aids the author of the module, who can use it to verify that the module works properly. However, the program can be useful to others as well.

A programmer who must use a library module that someone else has written needs to know in detail what the module does. If the module is not properly commented or lacks complete documentation, the programmer may need to test the module's behavior. (This situation arises more often than it should.)

Other characteristics of a library module that may testing include:

- *error handling*: what happens if we try to take the square root of a negative number?
- *performance*: how long does the `Sin` function take to compute the sine of a number?
- *accuracy*: how accurate are the algorithms that the functions in `MATHLIB` use?

The `TestMATHLIB` program allows us to test six of the functions exported by `MATHLIB`. The user of the program enters (1) a one-letter code indicating which function to test and (2) a real number to which the function should be applied; the program calls the selected function and prints the result. This sequence of events repeats indefinitely to allow the user to test more than one function or to test the same function on different values. The program terminates when the user presses the escape key.

The `LOOP` statement is ideal for this program because the test for termination (did the user press the escape key?) naturally falls in the middle of the loop. Nested inside the `LOOP` statement is a `CASE` statement that recognizes the six one-letter commands that the user may enter. The `CASE` statement's `ELSE` clause prints an error message when the user types a character other than one of the six allowable commands. Using `CAP` in the case expression allows the user to enter commands as either uppercase or lowercase letters.

```
MODULE TestMATHLIB;
  (* tests the MATHLIB module *)

  FROM IO IMPORT RdKey, WrChar, WrStr, WrLn, RdLngReal, WrLngReal;
  FROM MATHLIB IMPORT ATan, Cos, Exp, Log, Sin, Sqrt;

  CONST esc = 33C;

  VAR command: CHAR;
      realNum: LONGREAL;

BEGIN
  WrStr("When prompted, enter a real number and a one-letter command");
  WrLn;
  WrStr("representing a mathematical function. Legal commands are:");
  WrLn;
```



```

WrStr("a (arctan), c (cosine), e (exp), l (natural log),");
WrLn;
WrStr("q (square root), and s (sine). Press Esc to exit.");
WrLn;
WrLn;

LOOP
  WrStr("Enter a one-letter command: ");
  command := RdKey();
  IF command = esc THEN EXIT END;
  WrChar(command);
  WrLn;

  WrStr("Enter a real number: ");
  realNum := RdLngReal();

  CASE CAP(command) OF
    'A': WrLngReal(ATan(realNum), 15, 1) |
    'C': WrLngReal(Cos(realNum), 15, 1) |
    'E': WrLngReal(Exp(realNum), 15, 1) |
    'L': WrLngReal(Log(realNum), 15, 1) |
    'Q': WrLngReal(Sqrt(realNum), 15, 1) |
    'S': WrLngReal(Sin(realNum), 15, 1)
  ELSE
    WrStr("Command not recognized.")
  END;
  WrLn;
END
END TestMATHLIB.

```

The FOR Statement

The FOR statement allows a sequence of statements to execute repeatedly as a progression of values is assigned to a certain variable. For example, to find the sum of the squares of the first 100 positive integers, we could write

```

sum := 0;
FOR i := 1 TO 100 DO
  sum := sum + i * i
END;

```

The statement `sum := sum + i * i` is executed 100 times, with `i` assuming the values 1, 2, ..., 100, in that order. The variable `i` is called the *control variable*, and the values 1 and 100 are called the *starting value* and *limit*, respectively.

The number of repetitions need not be a constant, as long as we can express it in terms of the values of program variables. A more general program would read a number `n`, then compute the sum of the squares of the first `n` positive integers:

```

sum := 0;
n := RdCard();
FOR i := 1 TO n DO
  sum := sum + i * i
END;

```


We don't know in advance how many times the loop will be executed, but it is fixed once the program has read the value of *n*. This example illustrates the difference between the **WHILE**, **REPEAT**, and **LOOP** statements (which perform *indefinite* iteration) and the **FOR** statement (which performs *definite* iteration). The number of times the body of a **FOR** statement is executed should not depend on the actions taking place inside the body; in general, this property does not hold for the **WHILE**, **REPEAT**, and **LOOP** statements.

As an option, a **FOR** statement may specify an *increment* for the control variable. For example, if we want to sum the squares of the odd positive integers up to *n*, we could write

```
sum := 0;
FOR i := 1 TO n BY 2 DO
    sum := sum + i * i
END;
```

Now *i* takes on the values 1, 3, 5, ..., *n* (or *n* - 1 if *n* is even). Thus, *i* is incremented by 2 each time through the loop, instead of 1. Modula-2 allows negative increments. If *n* is odd, the following **FOR** statement will give the same answer as the previous one:

```
FOR i := n TO 1 BY -2 DO
    sum := sum + i * i
END;
```

Modula-2 restricts the use of the **FOR** statement in a number of ways:

- The type of the control variable must be a basic type (other than **REAL** or **LONG-REAL**), an enumeration type, or a subrange type. Control variables of type **CHAR** or an enumeration type are often useful. For example, the following **FOR** statement prints a table showing the ordinal numbers of the characters between 'A' and 'Z':

```
FOR letter := 'A' TO 'Z' DO
    WrCard(ORD(letter), 1);
    WrStr(" ");
    WrChar(letter);
    WrLn
END;
```
- The increment, if present, must be a constant expression of a whole number type (regardless of the type of the control variable).
- The types of the starting value and limit must be compatible with the type of the control variable.
- The control variable cannot be
 - a component of a structured variable (an array element or a record field)
 - an anonymous variable named by dereferencing a pointer
 - a variable imported from another module
 - a formal parameter
- The body of a **FOR** statement should not change the control variable nor should it alter the starting value or limit.

- Once a **FOR** statement has terminated, the value of its control variable is undefined. ("Undefined" means that the value may vary from one Modula-2 system to another and cannot be depended on.)

If a **FOR** statement whose increment is positive has a starting value greater than its limit, the body of the loop is not executed even once. (The same is true if the increment is negative and the starting value is less than the limit.) Watch out for the following error, however:

```
VAR i, n: CARDINAL;
.
.
.
FOR i := 0 TO n - 1 DO
.
.
.
END;
```

Suppose that the value of **n** is 0 at the time program control reaches the **FOR** statement. Instead of the body of the **FOR** loop being skipped, an error occurs: the expression **n - 1** has type **CARDINAL**, but subtracting 1 from 0 gives a non-**CARDINAL** result. We can avoid the problem by declaring **i** and **n** to have type **INTEGER** or by putting the **FOR** statement inside an **IF** statement that tests whether **n** is greater than 0 before executing the **FOR** statement.

Procedure Declarations

Procedures, like other entities in a Modula-2 program, must be declared. A procedure declarator specifies

- the name of the procedure
- information about the procedure's parameters
- the type of value that the procedure returns (if it is a function)
- declarations of entities that belong to the procedure
- a sequence of statements to be executed when the procedure is called

Chapter 5

Procedures

A *procedure* is an entity that a program can call (activate) to perform a computation. We can divide procedures into two groups: (1) *functions*, which compute a single value and whose calls appear in expressions; and (2) *proper procedures*, whose calls are statements.

In earlier chapters we used standard procedures and procedures imported from library modules. The standard procedure **ABS**, for example, is a function. When given a number x , **ABS** computes the absolute value of x ; a call of **ABS** appears in an expression, representing the value that **ABS** computes. The standard procedure **INC**, however, is a proper procedure. A call of **INC** is a statement; executing **INC** changes the value stored in a variable.

The rest of this chapter discusses Modula-2's rules for using procedures. The first section presents an overview of how programmers declare and call procedures, and discusses the differences between local and global entities. The second section shows how to write procedures with parameters, while the last section covers functions.

Procedure Declarations

Procedures, like other entities in a Modula-2 program, must be declared. A *procedure declaration* specifies

- the name of the procedure
- information about the procedure's parameters
- the type of value that the procedure returns (if it is a function)
- declarations of entities that belong to the procedure
- a sequence of statements to be executed when the procedure is called

The simplest procedures require no parameters. For example, the following declaration is for a procedure that a program can call when it encounters a fatal error:

```
PROCEDURE FatalError;
  (* writes an error message and terminates program *)
BEGIN
  WrStr("Fatal error; abandoning program.");
  WrLn;
  HALT
END FatalError;
```

More often, however, a procedure has one or more parameters. The following procedure exchanges the contents of two **INTEGER** variables supplied as parameters:

```
PROCEDURE Swap(VAR i, j: INTEGER);
  (* exchanges the contents of i and j *)
  VAR temp: INTEGER;
BEGIN
  temp := i;
  i := j;
  j := temp
END Swap;
```

The word **VAR** preceding the parameters **i** and **j** indicates that **i** and **j** represent variables (not expressions) to supply when calling **Swap**. **Swap** contains a declaration of a variable called **temp** the procedure uses for temporary storage during the exchange of **i** and **j**.

Both **FatalError** and **Swap** are proper procedures, not functions. We discuss functions later in this chapter; however, much information in this section pertains to functions as well as to proper procedures.

A procedure declaration has two parts: (1) the *procedure heading*, which gives information necessary to call the procedure, and (2) the *procedure body*. A procedure body contains declarations of entities belonging to the procedure and statements to be executed when the procedure is called. The reserved word **BEGIN** precedes the statements; the reserved word **END** follows them.

The heading of a procedure consists of the reserved word **PROCEDURE**, the name of the procedure, and (optionally) a list of formal parameters. Formal parameters are identifiers that represent variables or expressions to supply when calling the procedure.

The name of a procedure must appear twice in its declaration: (1) in the heading (after the word **PROCEDURE**) and (2) at the end (after the word **END**). The requirement that the name appear at the end of the declaration helps make programs more readable, because the name clearly marks the end of a procedure declaration and makes it easy to find. This rule also enables a Modula-2 compiler to recover from an error caused by an unpaired **BEGIN** or **END**.

Procedure Calls

A *procedure call* consists of a procedure name and (if the procedure has parameters) a list of *actual parameters*. The call of a proper procedure is a statement (function calls are expressions). For example, a call of **Swap** looks like this:

```
Swap(lower, upper);
```

When program control reaches this statement, **Swap** begins to execute. When the last statement in the body of **Swap** has been executed, the procedure *returns*; the next statement to be executed is the statement after the call of **Swap**.

A call of a proper procedure with no parameters consists of the procedure name alone:

```
FatalError;
```

The programmer may put a pair of empty parentheses after the name, if desired:

```
FatalError();
```

As a language extension, TopSpeed Modula-2 allows a procedure with two parameters to be called using infix notation. The name of the procedure is enclosed between \ symbols and placed between its actual parameters. For example, the call of **Swap** given earlier could be written

```
lower \Swap\ upper;
```

Declarations of Local Entities

A procedure declaration can itself include declarations. Typically, these declarations are of constants, types, or variables, but they can also be declarations of procedures or even modules. Entities declared within a procedure are *local* to the procedure: they exist only when the procedure is being executed and their names are visible only inside the procedure. Entities referenced in a procedure but not declared in the procedure are *global* to the procedure.

To see the difference, consider the following procedure:

```
PROCEDURE WriteLines(n: CARDINAL);
  (* writes n blank lines *)
BEGIN
  FOR i := 1 TO n DO
    Writeln
  END
END WriteLines;
```

Because **i** is not declared in **WriteLines**, it is global to the procedure. Each call of **WriteLines** changes the value of **i**, overwriting the value stored in **i** prior to the call.

We call this change to the value of *i* a *side effect*. A side effect is a hidden—and possibly unintentional—effect that results from a procedure call.

Side effects are usually unwanted. A programmer using `WriteLines` has no reason to suspect that it does anything but write blank lines; the change to *i* will come as a surprise. The programmer should be able to view `WriteLines` as an abstraction; it should not be necessary to read the statements inside `WriteLines` in order to use it.

Programmers can often avoid undesirable side effects by using local variables instead of global variables. By declaring *i* as a local variable within `WriteLines`, we can ensure that the procedure does not modify any global variables as a side effect:

```
PROCEDURE WriteLines(n: CARDINAL);
  (* writes n blank lines *)
  VAR i: CARDINAL;
BEGIN
  FOR i := 1 TO n DO
    WrLn
  END
END WriteLines;
```

Storage for a procedure's local variables is allocated (on a run-time stack) when the procedure is called. When the procedure returns, this storage is released (by popping the stack), thereby becoming available for subsequent procedure calls. Consequently, the values of its local variables are lost between calls of the procedure.

The limited lifetime of local variables has advantages and disadvantages. The chief advantage is that procedures allow efficient use of storage; by making variables local instead of global wherever possible, a programmer can minimize the amount of data storage that a program needs. The primary disadvantage is that programs cannot have variables that are local to a procedure yet retain their values between calls of the procedure (these are often called "own" variables). If a program needs to save the value of a local variable between calls of a procedure, the programmer must declare the variable global to the procedure or supply it as a parameter to the procedure. In either case, however, the variable is visible outside the procedure. One use of local modules (Chapter 9) is for hiding a procedure's "own" variables from the rest of the program.

The RETURN Statement

A proper procedure returns when the last statement in its body has been executed. A procedure may terminate prematurely by executing a `RETURN` statement. For example, the following procedure searches a line of input for a particular character; when it finds the character, the procedure immediately returns.

```
PROCEDURE SearchForChar(ch: CHAR);
  (* read characters until ch is found or end of line is
     reached; issue an error message in the latter case *)
  VAR inputChar: CHAR;
```



```
BEGIN
  REPEAT
    inputChar := RdKey();
    IF inputChar = ch THEN RETURN END
  UNTIL inputChar = 15C;
  WrStr("Search failed; character not found");
  WrLn
END SearchForChar;
```

If a **RETURN** statement is the only statement appearing inside an **IF** statement, we usually write the **IF** statement on a single line; we adopted a similar convention for the **EXIT** statement.

In a function, the **RETURN** statement serves a second purpose: In addition to causing the function to terminate, a **RETURN** statement indicates the value that the function should return. In a function, an expression (the value to return) must follow the word **RETURN**; in a proper procedure, an expression must *not* follow the word **RETURN**. It is an error for a function to "fall through" without executing a **RETURN** statement; in a proper procedure, using **RETURN** statements is optional.

Order of Procedure Declarations

According to Wirth's *Programming in Modula-2*, procedure declarations may occur in any order; thus, a program may call a procedure before its declaration has appeared. However, a number of recent Modula-2 compilers (including TopSpeed Modula-2) make only one pass over a program; as a result, they do not allow a procedure call to precede the declaration of the procedure.

When using a one-pass compiler, it is best to arrange procedures to avoid forward references. Such an arrangement is not always possible, however, when using *recursive* procedures. A procedure **P** can be recursive in two ways: (1) the body of **P** may contain a call of **P** (*direct recursion*), or (2) **P** may call **Q**, which calls **R**, and so on, until **P** is called (*indirect recursion*). For example, the following procedures exhibit a simple form of indirect recursion:

```
PROCEDURE P(i: INTEGER);
BEGIN
  (* body contains a call of Q *)
END P;

PROCEDURE Q(j: INTEGER);
BEGIN
  (* body contains a call of P *)
END Q;
```

Direct recursion causes no problems for a one-pass compiler, but indirect recursion does. In the example, the call of **Q** (in the body of **P**) precedes the declaration of **Q**. But if we declare **Q** first, then the call of **P** precedes the declaration of **P**. To resolve this problem, TopSpeed Modula-2 provides a special form of procedure declaration in which the word

FORWARD replaces the body of the procedure (the full heading must appear, however). This declaration signals the compiler that a complete declaration of the procedure appears later in the same block; in the meantime, this "preliminary" declaration provides information that the compiler needs to process calls of the procedure. The following declarations could handle our example of indirect recursion:

```
PROCEDURE Q(j: INTEGER); FORWARD;
```

```
PROCEDURE P(i: INTEGER);
BEGIN
  (* body contains a call of Q *)
END P;
```

```
PROCEDURE Q(j: INTEGER);
BEGIN
  (* body contains a call of P *)
END Q;
```

TopSpeed Modula-2 requires that the full declaration repeat the procedure's parameter list and—if it is a function—the result type.

Parameters

Most procedures depend on data that varies from one call to the next. For this reason, Modula-2 allows a procedure heading to include a list of identifiers that represent variables or expressions to supply when calling the procedure. The programmer can use these identifiers, known as *formal parameters*, in the body of the procedure in the same fashion as ordinary variables.

The procedure **SolveQuadratic** demonstrates the usefulness of parameters. **SolveQuadratic** computes the roots of the equation

$$ax^2 + bx + c = 0$$

assuming that $a \neq 0$ (otherwise there is only one root) and $b^2 - 4ac \geq 0$ (otherwise the roots are imaginary). **SolveQuadratic** needs five parameters: three for input (**a**, **b**, and **c**) and two for output (**root1** and **root2**).

```
PROCEDURE SolveQuadratic(a, b, c: REAL; VAR root1, root2: REAL);
  (* finds both roots of the quadratic equation  $ax^2 + bx + c = 0$ ,
     assuming that  $a \neq 0$  and  $b^2 - 4ac \geq 0$  *)
  VAR squareRoot, denominator: REAL;
BEGIN
  squareRoot := Sqrt(b * b - 4.0 * a * c);
  denominator := 2.0 * a;
  root1 := (-b + squareRoot) / denominator;
  root2 := (-b - squareRoot) / denominator;
END SolveQuadratic;
```

A call of a procedure with parameters must include a list of *actual parameters*. The number of actual parameters must be the same as the number of formal parameters. Cor-

correspondence between actual and formal parameters is by position: the first actual parameter corresponds to the first formal parameter, the second actual parameter to the second formal parameter, and so on. The type of each actual parameter must match the type of the corresponding formal parameter (the meaning of "match" depends on whether the formal parameter is a variable parameter or a value parameter, as we will see later). A call of `SolveQuadratic` might look like this:

```
SolveQuadratic(1.0, -8.0, 15.0, r1, r2);
```

The actual parameters are the expressions `1.0`, `-8.0`, and `15.0` and the variables `r1` and `r2`; they match the formal parameters `a`, `b`, `c`, `root1`, and `root2`, in that order.

Modula-2 provides two kinds of formal parameters: *variable parameters* and *value parameters*. In a procedure heading, if the reserved word `VAR` precedes a list of formal parameters, then they are variable parameters; otherwise, they are value parameters. Any changes made to a variable parameter within the body of the procedure also affect the corresponding actual parameter, which must be a variable (more precisely, a designator). However, if a formal parameter is a value parameter, the corresponding actual parameter—which may be any form of expression—is protected against change. Variable parameters allow information to flow into and/or out of a procedure; value parameters allow information to flow into a procedure, but not out of the procedure.

In the procedure `SolveQuadratic`, we made `a`, `b`, and `c` value parameters, because the procedure needs only the values of the actual parameters. We could have declared `a`, `b`, and `c` to be variable parameters instead, because `SolveQuadratic` does not change their values. Making them value parameters is preferable, because doing so documents the fact that `SolveQuadratic` needs only the values of the matching actual parameters. Furthermore, making `a`, `b`, and `c` variable parameters would prove too restrictive, because the corresponding actual parameters would have to be variables; we could not use the expressions `1.0`, `-8.0`, and `15.0` as actual parameters.

We have declared `root1` and `root2` to be variable parameters, however. By assigning values to `root1` and `root2`, `SolveQuadratic` can change the values of the corresponding actual parameters; these values are then available for use after the procedure returns. For example, after the call of `SolveQuadratic` shown earlier, the values of `r1` and `r2` are 5 and 3, respectively. Declaring `root1` and `root2` to be value parameters would be a mistake, because assignments to `root1` and `root2` would not affect `r1` and `r2`.

The type of a formal parameter is usually given as a qualified identifier (the name of a type). If the words `ARRAY OF` precede the type name, the parameter is an *open array parameter*. If a procedure has an open array parameter, the corresponding actual parameter can be an array of any length, provided that its component type matches the type appearing after the words `ARRAY OF`. We consider open array parameters in Chapter 7.

Variable Parameters

The actual parameter corresponding to a variable parameter must be a variable, an array component, a record component, or an anonymous variable accessed through a pointer. It cannot be a constant or any other form of expression, even if no assignments to the formal parameter appear in the procedure body.

A variable parameter and each corresponding actual parameter must have the same type. Modula-2 allows exceptions to this rule for parameters of types **WORD** and **ADDRESS** (see Chapter 10).

Value Parameters

The actual parameter corresponding to a value parameter can be any kind of expression (including, but not limited to, constants and variables). The type of the actual parameter must be assignment-compatible with the type of the value parameter, unless the value parameter has type **WORD** or **ADDRESS**, in which case special rules apply. The assignment compatibility requirement allows the types of formal and actual parameters to differ in some cases; in particular, a formal parameter may have type **INTEGER** while the matching actual parameter has type **CARDINAL**, or vice-versa.

When a procedure with a value parameter is called, the corresponding actual parameter is evaluated and its value copied to the formal parameter, which serves as a local variable. The procedure can change the value of the formal parameter by assignment, but any change will not affect the actual parameter, even if it is a variable. For example, suppose that we rewrite the procedure **WriteLines** to use a **WHILE** statement instead of a **FOR** statement:

```
PROCEDURE WriteLines(n: CARDINAL);
  (* writes n blank lines *)
BEGIN
  WHILE n > 0 DO
    WrLn;
    DEC(n)
  END
END WriteLines;
```

WriteLines changes the value of **n** by decrementing it each time through the body of the **WHILE** statement. But because **n** is a value parameter, only the procedure's local copy of **n** changes; the actual parameter does not. A value parameter behaves exactly like a local variable; the only difference between the two is that a value parameter has an initial value (the value of the matching actual parameter).

So that changes to a value parameter will not affect the corresponding actual parameter, a Modula-2 system must copy the actual parameter each time the procedure is called. If the actual parameter requires a large amount of storage (for example, if it is a large ar-

ray), this copying can be expensive in terms of both time and space. Variable parameters incur no such penalty.

At least two other factors also affect the choice between variable and value parameters. First, if the actual parameter should reflect changes to the formal parameter, the latter must be a variable parameter. Second, if the actual parameter will be an expression other than a variable, we must use a value parameter. If neither condition holds, a value parameter is probably the best choice, unless efficiency dictates using a variable parameter.

Functions

Every Modula-2 procedure is either a *function* or a *proper procedure*. A function computes a single value; furthermore, calls of the function procedure may appear in expressions. For example, the expression on the right side of the following assignment contains call of the **Sin** and **Cos** functions:

```
tanx = Sin(x) / Cos(x);
```

A call of a proper procedure, however, is a statement. A proper procedure must store its results into actual parameters or global variables; a call of a proper procedure cannot appear in an expression.

A function declaration differs only slightly from the declaration of a proper procedure. First, the function's heading must specify the type of value it returns. Second, the body of the function must include a **RETURN** statement that, when executed, indicates the value that the function returns. The following declaration of the **SquareRoot** function illustrates these differences:

```
PROCEDURE SquareRoot(n: CARDINAL): CARDINAL;
  (* returns the greatest integer whose square is less than
     or equal to n *)
  VAR root: CARDINAL;
BEGIN
  root := 1;
  WHILE root * root <= n DO
    INC(root)
  END;
  RETURN root - 1
END SquareRoot;
```

The appearance of a colon following **SquareRoot**'s parameter list indicates that **SquareRoot** is a function rather than a proper procedure; the type name **CARDINAL** after the colon specifies that **SquareRoot** returns a value of type **CARDINAL**. The **RETURN** statement indicates that **SquareRoot** returns the final value of the expression **root - 1**.

Any expression can appear after the word **RETURN**, provided that its type is assignment-compatible with the type specified in the heading of the function (**CARDINAL**, in this

case). The body of a function may contain more than one **RETURN** statement; the first one encountered during the execution of the function causes it to return. Reaching the end of a function body without executing a **RETURN** statement is an error.

A function call consists of a designator followed by a list of actual parameters enclosed in parentheses:

```
SquareRoot(a * a + b * b)
```

A proper procedure call is a statement, but a function call is an expression of the type specified in the function heading. Consequently, a function call can appear wherever a value of that type can (only calls of standard functions may appear in constant expressions, however). Some examples:

```
VAR i, j, sum: CARDINAL;
.
.
.
sum := SquareRoot(i) + SquareRoot(j);
WrCard(SquareRoot(i + j), 1);
```

In TopSpeed Modula-2, functions can return values of any type. However, the syntax of Modula-2 prohibits using the value returned by a function as a designator. Thus a program cannot

- select a component of an array or record result
- dereference a pointer result
- call a procedure result

This restriction should cause no problems: a program can store the value returned by a function in a variable, then select, dereference, or call the variable.

Functions, like proper procedures, can be recursive. The following function computes the *greatest common divisor* of two **CARDINAL** values *i* and *j*: the largest integer that divides both *i* and *j* evenly.

```
PROCEDURE GCD(i, j: CARDINAL): CARDINAL;
  (* returns the greatest common divisor of i and j *)
BEGIN
  IF j = 0 THEN
    RETURN i
  ELSE
    RETURN GCD(j, i MOD j)
  END
END GCD;
```

A function must always have a parameter list, but the list may be empty. Although functions with no parameters are not common, they are occasionally useful. For example, we may need a function that will generate a sequence of pseudorandom **CARDINAL** numbers

$c_1, c_2, \dots$ using the linear congruence method. In this method, we generate the $(i+1)$ st number in the sequence from the i th number by the formula

$$c_{i+1} = (\text{multiplier} \cdot c_i + \text{increment}) \bmod \text{modulus}$$

where *multiplier*, *increment*, and *modulus* are appropriate constants. Each call of **Random** returns the next number in the sequence. **Random** also saves this number in a global variable **randomCardinal** so that it can generate the next number in the sequence at the next call of **Random**. Because the value returned by **Random** depends only on the value stored in **randomCardinal**, **Random** does not need any parameters:

```
PROCEDURE Random(): CARDINAL;
  (* returns a pseudorandom number between 0 and modulus - 1;
   * modifies the variable randomCardinal *)
BEGIN
  randomCardinal := (multiplier * randomCardinal + increment) MOD modulus;
  RETURN randomCardinal
END Random;
```

The value returned by **Random** can be divided by **modulus** to give a pseudorandom real number between 0 and 1.

In a call of a function with no parameters, an empty list of actual parameters must follow the name of the function:

```
rand := Random();
```

Modula-2 requires the empty parentheses because the language allows procedures to be treated as data (see Chapter 8). A Modula-2 compiler would interpret the statement

```
rand := Random;
```

as an assignment of the **Random** function itself to the variable **rand**.

As a language extension, TopSpeed Modula-2 allows a function with two parameters to be called using infix notation. For example, consider the function **Pow** exported by the **MATHLIB** module. Instead of writing

```
w := Pow(x, Pow(y, z));
```

we could write

```
w := x \Pow\ (y \Pow\ z);
```

Functions called in this fashion are treated as operators with precedence lower than that of any normal Modula-2 operator.

Chapter 6

Enumeration and Subrange Types

Chapter 3 described the basic types of Modula-2. This chapter and the next two show how to augment this collection by defining new types. This chapter begins with an overview of Modula-2's mechanisms for defining new types. Later sections then explain the rules for defining enumeration and subrange types. The final section presents Modula-2's type-checking rules. Chapters 7 and 8 describe other methods for specifying new types.

Programmer-defined Types

Some Modula-2 types are predefined; a Modula-2 compiler recognizes these *standard types* without the programmer having to define them. The basic types (Chapter 3) are all standard; the only other standard types are **PROC** (Chapter 8) and **BITSET** (Chapter 10). In addition to using standard types, Modula-2 programmers can create new types.

Every Modula-2 type, whether standard or not, falls into one of four categories; it is either an *unstructured*, *structured*, *pointer*, or *procedure* type.

A value of an unstructured type is a single, indivisible item of data, such as a number or character. The basic types are all unstructured. Modula-2 also allows programmers to create new unstructured types in two ways: (1) by listing (enumerating) the values of the new type or (2) by selecting a sequence of consecutive values from a previously-defined unstructured type. We call these *enumeration types* and *subrange types*, respectively.

A value of a structured type is a collection of data items. The only standard structured type is **BITSET**; the programmer must define all other structured types. Structured types are of three kinds: *array types*, *record types*, and *set types*. When creating a new structured type, the programmer must specify how many components a value of the type will

contain, what the types of these components will be, and how to access the components. Chapter 7 discusses structured types.

Pointer and procedure types are special-purpose types that share few of the properties of the other types. A value of a pointer type is a reference to another data value; a value of a procedure type is a reference to a procedure. Chapter 8 discusses pointer and procedure types.

Modula-2 types, both standard and programmer-defined, can be summarized as follows:

- unstructured types
 - standard (**CARDINAL**, **INTEGER**, **LONGINT**, **REAL**, **LONGREAL**, **BOOLEAN**, **CHAR**)
 - enumeration
 - subrange
- structured types
 - array
 - record
 - set (including the standard type **BITSET**)
- pointer types
- procedure types (including the standard type **PROC**)

TopSpeed Modula-2 provides additional standard types as language extensions, including **SHORTCARD**, **LONGCARD**, **SHORTINT**, and several low-level types (see Chapter 10).

Type Specifications

A *type specification* is a description of a new type. A type specification must contain enough information for a Modula-2 compiler to determine what values belong to the type.

To illustrate simple forms of type specification, we will use enumeration and subrange types (the subjects of this chapter). An enumeration type is an ordered sequence of values represented by identifiers. The specification of an enumeration type is a list of identifiers enclosed in parentheses:

```
(small, medium, large)
```

A subrange type is a sequence of consecutive values chosen from an existing unstructured type. The specification of a subrange type is a pair of constant expressions separated by two periods and enclosed in brackets:

```
[32..212]
```


Both forms of specification inform the compiler what the values of the new type are. The first specification indicates that the type contains three (and only three) values, represented by the identifiers **small**, **medium**, and **large**. The second specification defines a type with 181 values (the integers 32, 33, 34, ..., 212).

Type specifications can appear in two places: variable declarations and type declarations. In a variable declaration, a type specification can appear in place of a type name:

```
VAR shirtSize: (small, medium, large);
```

When a type specification appears in a variable declaration, the variable's type is *anonymous*, because the programmer has created a new type without naming it. The type of the variable **shirtSize** is anonymous.

Type Declarations

A programmer can name a new type by using a *type declaration*, then use the name in the same way as the name of a standard type. For example, instead of using a type specification to declare the variable **shirtSize**, we could first declare a type **Sizes**, then use **Sizes** to declare **shirtSize**:

```
TYPE Sizes = (small, medium, large);
VAR shirtSize: Sizes;
```

The name of a new type may appear in any number of variable declarations:

```
VAR hatSize: Sizes;
    coatSize: Sizes;
```

A type declaration consists of a type name (chosen by the programmer), an equals sign, and a type specification. Type declarations appear in sequences of one or more. The reserved word **TYPE** must precede the first type declaration in a sequence; in addition, a semicolon must terminate every type declaration:

```
TYPE Sizes = (small, medium, large);
    Temperatures = [32..212];
```

Once we have declared a new type, we can use it to declare other types:

```
TYPE Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
    Weekdays = [Mon..Fri];
```

The type **Weekdays** is a subrange of the **Days** type. The order of these two type declarations is important, because Modula-2 does not allow forward references in declarations. (This rule has one exception: the specification of a pointer type may refer to a type whose declaration has not yet appeared.)

When creating a new type, the programmer must decide whether to give the type a name (using a type declaration) or to leave it anonymous. The former is usually preferable, for several reasons:

- A type name can appear many times in a program, simplifying the program and providing a form of documentation—as long as the type name is meaningful. Furthermore, the program is easier to modify. (Suppose, for example, that we decide to change the lower or upper bound of the **Temperatures** subrange. We need only modify the declaration of **Temperatures**, not the declarations of variables of type **Temperatures**.)
- An anonymous type differs from all other types (either named or anonymous) in a program. Consequently, Modula-2's type-checking rules severely limit the usefulness of a variable whose type is anonymous, as we will see later in this chapter.
- The programmer must use a name—not a specification—to declare the type of a formal parameter.

Enumeration Types

Often a programmer faces the need to declare a variable that will take on one of a small set of values. For example, a program to process astronomical observations might require a variable whose values represent the nine planets in the solar system. One possibility is to declare the variable to be of type **CARDINAL** and to use a code for each planet:

```
CONST Mercury = 0;
      Venus  = 1;
      Earth  = 2;
      Mars   = 3;
      Jupiter = 4;
      Saturn = 5;
      Uranus  = 6;
      Neptune = 7;
      Pluto   = 8;
VAR planet1, planet2: CARDINAL;
CONST closestToEarth = Venus;
```

Modula-2 provides a better alternative: to declare an *enumeration type* with nine values, representing the nine planets.

```
TYPE Planets = (Mercury, Venus, Earth, Mars, Jupiter, Saturn,
               Uranus, Neptune, Pluto);
VAR planet1, planet2: Planets;
CONST closestToEarth = Venus;
```

Creating an enumeration type rather than using a code has several advantages. First, an enumeration type is easier to declare than the values of a code. Second, defining an enumeration type avoids the risk that two code names might accidentally be given the

same value. Most importantly, when the programmer uses code values (of type **CARDINAL**, say), the compiler cannot check that values are assigned only to variables of the right type. For example, suppose that values of type **CARDINAL** represent the planets, as sketched earlier. After the declarations

```
CONST Sun = 0;
      .
      .
      Sat = 6;
VAR Day: CARDINAL; (* value should be one of Sun, ..., Sat *)
```

the programmer might forget the name of the constant representing the planet Saturn and write

```
planet1 := Sat;
```

This statement is perfectly legal from Modula-2's point of view; only when the program produces the wrong results will the programmer become aware of any problem. Trying to track down an error of this nature can be difficult; using enumeration types allows the compiler to detect the error.

To define an enumeration type, the programmer simply invents names for the values of the type, then encloses a list of these names in parentheses. The names in the list must all be different; furthermore, no name may be the same as the name of another entity declared in the same block. If two enumeration types are declared in the same block, no value belonging to both can have the same name. The following declarations are illegal, because the identifiers **red** and **blue** appear in two enumeration types:

```
TYPE FlagColors = (red, white, blue);
   PrimaryColors = (blue, red, yellow); (* illegal *)
```

The basic type **BOOLEAN** is essentially an enumeration type defined as follows:

```
TYPE BOOLEAN = (FALSE, TRUE);
```

BOOLEAN has special properties not shared by programmer-defined enumeration types, however.

Enumeration types have numerous uses. The type of a tag field in a record with a variant part (Chapter 7) is often an enumeration. Enumeration types are useful for defining "state" variables—variables that record the current state of a program. For example, the following variable allows a program to keep track of whether it is adding entries to a table or retrieving them from the table:

```
VAR mode: (enterMode, fetchMode);
```

Enumeration types are also useful for expressing a result computed by a function:


```
TYPE CompareResult = (less, equal, greater);
PROCEDURE CompareWord(w1, w2: Word): CompareResult;
```

or supplying information to a procedure:

```
TYPE EchoMode = (echo, noEcho);
PROCEDURE ReadChar(VAR ch: CHAR; mode: EchoMode);
```

This use deserves special mention. Many procedures have parameters that can take on only two values; the programmer may feel tempted to declare such a parameter to have type **BOOLEAN**. However, using a two-valued enumeration type provides better program documentation. The call

```
ReadChar(ch, noEcho);
```

is much more understandable than the call

```
ReadChar(ch, FALSE);
```

The ORD and VAL Functions

In the specification of an enumeration type, the order in which the identifiers (enumeration values) appear is important. Each value has an *ordinal number* representing its position in the list. The first value in a list has ordinal number 0, the second has ordinal number 1, and so forth. The standard function **ORD** returns the ordinal number of an enumeration value:

<i>Expression</i>	<i>Value</i>
ORD (Mercury)	0
ORD (Mars)	3
ORD (Pluto)	8

The standard function **VAL** serves as an inverse to **ORD**. When given a type name **T** and a **CARDINAL** number **i**, **VAL** returns the value of type **T** whose ordinal number is **i**.

<i>Expression</i>	<i>Value</i>
VAL (Planets, 0)	Mercury
VAL (Planets, 3)	Mars
VAL (Planets, 8)	Pluto

Modula-2 provides no operations for enumeration types other than assignment and the relational operations:

```
planet2 := planet1;
planet1 := Mars;
IF planet2 < Earth THEN
  WrStr("Planet 2 is closer to the sun than Earth")
END;
```


Comparisons between values of an enumeration type depend on their ordinal numbers; for example, **Venus** is less than **Earth** because **ORD(Venus)** is less than **ORD(Earth)**.

The INC and DEC Procedures

Because no arithmetic operations are defined for enumeration types, a program cannot increment a variable of an enumeration type by an assignment statement of the following form:

```
planet1 := planet1 + 1;    (* illegal *)
```

Using **ORD** and **VAL** yields a somewhat complicated assignment:

```
planet1 := VAL(Planets, ORD(planet1) + 1);
```

Fortunately, the standard procedures **INC** and **DEC** allow parameters of an enumeration type:

```
INC(planet1);
```

INC and **DEC** permit a second parameter that serves as a increment (or decrement). Thus, after the statements

```
planet1 := Earth;
DEC(planet1, 2);
```

are executed, the value of **planet1** is **Mercury**.

INC and **DEC** provide a convenient way to loop over all the values of an enumeration type (or a range thereof):

```
planet1 := Mercury;
WHILE planet1 <= Pluto DO
    .
    .
    .
    INC(planet1)
END;
```

Alternatively, we can use a **FOR** statement for this purpose:

```
FOR planet1 := Mercury TO Pluto DO
    .
    .
    .
END;
```


Calls of **INC** and **DEC** may produce undefined results: **INC(planet1)** is illegal if the value of **planet1** is **Pluto**; similarly, **DEC(planet1)** is illegal if the value of **planet1** is **Mercury**.

Example: Writing Values of an Enumeration Type

Because the programmer creates enumeration types, the programmer is also responsible for providing procedures to read and write enumeration values. Fortunately, I/O procedures for enumeration types are relatively easy to write.

To write enumeration values in human-readable form, a program may use a **CASE** statement to select a string representing a particular value. For example, the following procedure writes values of type **Planets**.

```
PROCEDURE WritePlanet(planet: Planets);
  (* writes the name of a value of type Planets *)
BEGIN
  CASE planet OF
    Mercury: WrStr("Mercury") |
    Venus:   WrStr("Venus")   |
    Earth:   WrStr("Earth")   |
    Mars:    WrStr("Mars")    |
    Jupiter: WrStr("Jupiter") |
    Saturn:  WrStr("Saturn")  |
    Uranus:  WrStr("Uranus")  |
    Neptune: WrStr("Neptune") |
    Pluto:   WrStr("Pluto")
  END
END WritePlanet;
```

As this example shows, **CASE** statements are ideal for choosing among the values of an enumeration type.

Alternatively, a program may need to write enumeration values to a file where another program will read them later. In this case, it is more efficient to write a code (typically the ordinal number of the enumeration value) rather than a string.

Subrange Types

Often the programmer knows in advance that a variable will assume values in a limited range within a particular type. For example, the variable **age**, representing a person's age, must have a value between 0 and 150, say. We can indicate these bounds by declaring the type of **age** to be a *subrange* of the type **CARDINAL**:

```
TYPE AgeRange = [0..150];  (* subrange of CARDINAL *)
VAR age: AgeRange;
```

There are three advantages to declaring **age** in this fashion instead of declaring it to be of type **CARDINAL**. First, the declaration helps a reader of the program understand what

age represents and what values it can assume. Second, the declaration may allow the compiler to reduce the amount of storage allocated for **age**. Third, the declaration allows the value of **age** to be checked as the program is executed: if an assignment to **age** would cause its value to be outside the specified range (indicating a programming error), the program will be terminated with an error message. (In TopSpeed Modula-2, this feature must be activated by the compiler directive **(*\$R+*)**.)

The values of a subrange type must be a range of consecutive values selected from a previously defined unstructured type, called the *base type*. The base type for a subrange must be a numeric type (other than **REAL** or **LONGREAL**), **CHAR**, or any enumeration type. Here are several subrange types:

```

TYPE UpperCase = ['A'..'Z'];      (* subrange of CHAR *)
    LowerCase = ['a'..'z'];      (* subrange of CHAR *)
    VHFChannels = [2..13];       (* subrange of CARDINAL *)
    CloseToSun = [Mercury..Mars]; (* subrange of Planets *)

```

The base type of a subrange is determined by the types of its bounds. Thus, the base type for **UpperCase** and **LowerCase** is **CHAR**, and the base type for **CloseToSun** is **Planets**. When the bounds are both integers, as is the case with **VHFChannels**, the base type is **INTEGER** if the smaller bound is negative and **CARDINAL** otherwise.

The specification of a subrange type consists of a pair of constant expressions, separated by the **..** symbol and enclosed in brackets. (In Modula-2, the **..** symbol separates the lower and upper limits of a range of values. We saw this use in the **CASE** statement.) The value of the first expression must be less than or equal to the value of the second (that is, the type must have at least one possible value). Optionally, a type name may precede the specification to indicate the base type of the subrange.

The ability to specify the base type of a subrange is useful when the base type would be **CARDINAL** by default but the programmer wants it to be **INTEGER** instead:

```

TYPE WaterTemperatures = INTEGER[32..212];

```

If the programmer omitted the type name **INTEGER**, the base type of **WaterTemperatures** would be **CARDINAL**. The choice of **INTEGER** rather than **CARDINAL** as the base type for **WaterTemperatures** is only important in situations where **WaterTemperatures** must be *compatible* with **INTEGER**. We return to this point later in the chapter.

A subrange type inherits all the operations of its base type. For example, all the operations legal for **CARDINAL** values also apply to values of type **AgeRange**, because **AgeRange** is a subrange of **CARDINAL**. If **age1**, **age2**, and **ageDifference** are variables of type **AgeRange**, Modula-2 allows the assignment

```

ageDifference := age1 - age2;

```


provided that the value of **age1 - age2** is not negative. In addition, Modula-2 allows the programmer to combine values of type **AgeRange** with values of type **CARDINAL** or another subrange of **CARDINAL**:

```
age1 := age2 + 10;
```

In general, Modula-2's operators require that their operands be *compatible*. For purposes of type compatibility, Modula-2 treats a value of a subrange type as though it belonged to the base type. (In the preceding statement, for example, the variable **age2** is treated as having type **CARDINAL**.) An arithmetic operation performed when either operand has a subrange type produces a value of the base type. Thus, the type of **age2 + 10** is **CARDINAL**.

Compatibility and Assignment Compatibility

Because Modula-2 is a strongly typed language, compilers must check the types of values appearing in Modula-2 expressions and statements. The kind of check performed varies, depending on the context. When a procedure is called, the type of each variable parameter must be the *same* as the type of the matching actual parameter. The operands in an expression need not be the same, but must be *compatible*, and the variable and expression in an assignment statement must be *assignment-compatible*. Thus, Modula-2 has three definitions of what is required for two types to "match." We can now give precise definitions of these three terms.

Type Equivalence

Two types are the *same* if they have the same name or if they were derived—directly or indirectly—from the same type name. Thus, the following types are all the same:

```
TYPE Length = REAL;
      Distance = Length;
      Mileage = Length;
      Mass = REAL;
```

For example, we can trace both **Distance** and **REAL** back to the type name **REAL**; we can trace **Distance** and **Mileage** back to **Length** (or **REAL**).

Types that merely have the same structure are *not* the same:

```
TYPE WorkDays = [Mon..Fri];
      SchoolDays = [Mon..Fri];
```

WorkDays and **SchoolDays** have the same base type and the same values, but we cannot trace them back to the same type name.

An anonymous type, by definition, cannot be the same as any other type. However, two variables with anonymous types can have the same type:

```
VAR day1, day2: [Mon..Fri];
```

Although both **day1** and **day2** have an anonymous type, their types are the same, because they appear in the same declaration. However, no other variables have the same type as **day1** and **day2**. If we declare **day3** as

```
VAR day3: [Mon..Fri];
```

then **day3**'s type is different from that of **day1** and **day2**.

In only one context must types must be the same, not merely compatible or assignment-compatible: The type of a variable parameter must be the same as the type of a corresponding actual parameter.

Compatibility

Two types T_1 and T_2 are *compatible* if they are the same, or if one is a subrange of the other, or if both are subranges with the same base type. Consider the following types:

```
TYPE SmallCards = [0..999];
      LargeCards = [100..9999];
```

SmallCards and **CARDINAL** are compatible, because the former is a subrange of the latter. **LargeCards** is compatible with **CARDINAL** for the same reason. Also, **SmallCards** and **LargeCards** are compatible, because both are subranges of **CARDINAL**.

Modula-2 requires compatibility for

- the operands in an arithmetic expression
- the operands in a relational expression
- the case labels and case expression in a **CASE** statement
- the starting value, limit, and control variable in a **FOR** statement

We now see the advantage of specifying a base type of **INTEGER** in the declaration

```
TYPE WaterTemperatures = INTEGER[32..212];
```

This declaration ensures that **WaterTemperatures** is compatible with **INTEGER** and all subranges of **INTEGER** (rather than compatible with **CARDINAL** and its subranges). Thus, **WaterTemperatures** values can be combined with other temperatures that may be negative:


```

VAR outsideTemperature: INTEGER;
    lakeTemperature: WaterTemperatures;
    .
    .
    .
IF outsideTemperature >= lakeTemperature THEN
    WrInt(outsideTemperature - lakeTemperature, 1);
    WrLn
END;

```

If **WaterTemperatures** were a subrange of **CARDINAL**, it would be illegal to compare **outsideTemperature** and **lakeTemperature** or to compute their difference.

Assignment Compatibility

Assignment statements require an even weaker compatibility condition. Two types are *assignment-compatible* if they are compatible or if both are **INTEGER** or **CARDINAL** or subranges with base types **INTEGER** or **CARDINAL**. Thus, the following types are assignment-compatible with each other and with **INTEGER** and **CARDINAL**:

```

TYPE SmallCards = [0..999];
    LargeCards = [100..9999];
    SmallInts = [-999..999];

```

In TopSpeed Modula-2, the types **SHORTCARD** and **SHORTINT** (and their subranges) are assignment-compatible, as are **LONGCARD** and **LONGINT** (and their subranges).

Modula-2 requires assignment compatibility for:

- the designator and expression in an assignment statement (with the exception of the assignment of a string literal to a string variable or variable of type **CHAR**)
- an array's index type and an expression used as an index
- a value parameter and each corresponding actual parameter
- the type of result to be returned by a function and the expression in a **RETURN** statement in the body of the function

Assignment compatibility does not guarantee the validity of an operation. If the variables **small** and **large** have types **SmallInts** and **LargeCards**, respectively, then the assignment

```
small := large;
```

is legal only when the value of **large** is between 100 and 999.

Summary

We can summarize Modula-2's three forms of type compatibility as follows:

- Types T_1 and T_2 are the *same*
if T_1 and T_2 have the same name
or if both types have been derived from the same type name by type declarations.
- Types T_1 and T_2 are *compatible*
if T_1 and T_2 are the same
or if one is a subrange of the other
or if both are subranges with the same base type
- Types T_1 and T_2 are *assignment-compatible*
if T_1 and T_2 are compatible
or if one is **CARDINAL** (or a subrange thereof) and the other is **INTEGER** (or a subrange thereof)

We have arranged these conditions in order of decreasing strength. Requiring compatibility is weaker than requiring types to be the same, because it allows a subrange to be treated (for type-checking purposes) as though it belonged to the base type. Assignment compatibility is even weaker, because it blurs the distinction between **INTEGER** and **CARDINAL**.

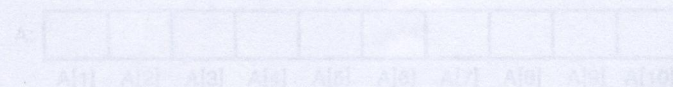
A final note: Special compatibility and assignment compatibility rules hold for the low-level types **ADDRESS**, **SHORTADDR**, **BYTE**, **WORD**, and **LONGWORD** provided by Top-Speed Modula-2. See Chapter 10 for details.

Array Types

An array is a structure containing a fixed number of values, all of the same type, that can be accessed by position. The values stored in an array are the components of the array; their type is the component type. The values that indicate the positions of components are indices; their type is the index type. For example, consider the following declaration:

```
TYPE ArrayType = ARRAY [1..10] OF INTEGER;
VAR A: ArrayType;
```

The component type of **A** is **INTEGER**, and the index type is **[1..10]** (a subrange of **CARDINAL**). We can visualize **A** as follows:



Chapter 7

Array, Record, and Set Types

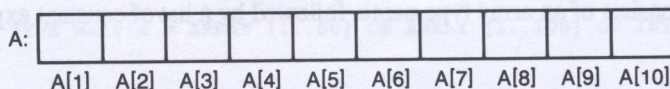
Chapter 6 showed how a programmer can create new unstructured types in Modula-2. Chapter 7 shows how to define new *structured* types—arrays, records, and sets. The first section covers both array types and the related topic of open array parameters, which make it possible to write procedures that can accept arrays of different sizes. The second section shows how to define and use string types; in Modula-2, strings are just one-dimensional arrays of characters. The third section discusses record types and the **WITH** statement, which simplifies the task of accessing fields in a record. The treatment of record types continues in the fourth section, which describes records with variant parts. The final section covers set types and the operations that Modula-2 provides for sets.

Array Types

An *array* is a structure containing a fixed number of values, all of the same type, that can be selected by position. The values stored in an array are the *components* of the array; their type is the *component type*. The values that indicate the positions of components are *indices*; their type is the *index type*. For example, consider the following declarations:

```
TYPE ArrayType = ARRAY [1..10] OF INTEGER;  
VAR A: ArrayType;
```

The component type of **A** is **INTEGER**, and the index type is **[1..10]** (a subrange of **CARDINAL**). We can visualize **A** as follows:



The name **A** refers to the array as a whole; the expressions **A[1]**, **A[2]**, ..., **A[10]** select individual components of **A**.

The components of an array may be of any type. The index type, however, must be **CHAR**, **BOOLEAN**, or an enumeration or subrange type. (As a language extension, TopSpeed Modula-2 allows the use of **CARDINAL** and **INTEGER**—not just their subranges—as index types. **SHORTCARD** and **SHORTINT** and their subranges are also legal index types in TopSpeed Modula-2, but **LONGCARD** and **LONGINT** and their subranges are not.) The following are all legal array types:

```
TYPE BitVector = ARRAY [0..255] OF BOOLEAN;
   CharCounts = ARRAY CHAR OF CARDINAL;
   LetterCounts = ARRAY ['a'..'z'] OF CARDINAL;
   PlanetValues = ARRAY Planets OF REAL;
               (* Planets was defined in Chapter 6 *)
```

The bounds of a subrange may be constant expressions, so type declarations such as the following are legal:

```
CONST range = 100;
TYPE Measurements = ARRAY [-(range-1)..range-1] OF REAL;
```

To select an array component, the programmer writes an expression in brackets after the name of the array. For example, if **A** is the array declared earlier, we can access the fifth component of **A** by writing any of the following: **A[5]**, **A[i]** (if **i** has the value 5), or **A[1 + 2 * j]** (if **j** has the value 2). Each of these is a *designator*; the expression inside the brackets is the *selector* or *index expression*. A designator of an array component has most of the properties of a variable; in particular, it can appear on the left side of an assignment statement:

```
FOR i := 1 TO 10 DO
  A[i] := 0    (* assign 0 to each component of A *)
END;
```

In an array designator, the type of the index expression must be assignment-compatible with the array's index type. (In the case of the array **A**, an index expression may have type **INTEGER**, **CARDINAL**, or any subrange of **INTEGER** or **CARDINAL**.) Furthermore, if the index type is a subrange, the value of the index expression must lie within the bounds of the subrange. Unless the index expression is constant, a Modula-2 compiler cannot check this condition; instead, Modula-2 systems perform the test as the program executes, terminating the program if the expression is out of range. (In TopSpeed Modula-2, index checking must be enabled by including the **(*\$I+*)** directive in the program.)

As a language extension, TopSpeed Modula-2 provides array *aggregates*. An array aggregate consists of an array type name, followed by a list of constant expressions, one for

each component of the array. Assigning an aggregate to an array changes all components of the array:

```
A := ArrayType(0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
```

Modula-2 allows references to whole arrays (as opposed to array components) only in certain contexts. First, an array name (representing a whole array) may appear on either side of an assignment statement:

```
A := B;
```

Such an assignment copies each component of **B** to the corresponding component of **A**. Second, the programmer may supply an array name as an actual parameter in a procedure call:

```
Sort(A);
```

In either case, the usual rules of type compatibility apply. For a programmer to assign one array to another, the arrays must have the same type. An array used as an actual parameter must have the same type as the corresponding formal parameter. (In the case of array types, the distinction between *same*, *compatible*, and *assignment-compatible* does not exist; all three terms are equivalent.)

As a language extension, TopSpeed Modula-2 also allows a program to compare two arrays of the same type for equality or inequality.

Using a type specification rather than a type name to declare an array limits the use of the array as a whole. For example, suppose that we use an array type specification to declare the variables **A** and **B**:

```
VAR A, B: ARRAY [1..10] OF INTEGER;
```

A and **B** have the same anonymous type. No other arrays can have the same type, so **A** and **B** can appear only in assignment statements in which one array is assigned to the other. **A** and **B** cannot appear as parameters in procedure calls (unless the corresponding formal parameter is an open array parameter, as we will see later). This form of declaration does not limit access to the components of **A** and **B**, however; the programmer can use them as **INTEGER** variables.

Multidimensional Arrays

In Modula-2, programmers create multidimensional arrays by nesting one-dimensional arrays. For example, the following declaration creates a 50-by-100 matrix type:

```
TYPE Matrix = ARRAY [1..50] OF ARRAY [1..100] OF INTEGER;
```


Because this sort of type declaration can be unwieldy, Modula-2 allows the equivalent declaration

```
TYPE Matrix = ARRAY [1..50], [1..100] OF INTEGER;
```

In general, the specification

$$\text{ARRAY } T_1, T_2, \dots, T_n \text{ OF } T$$

is an abbreviation for

$$\text{ARRAY } T_1 \text{ OF ARRAY } T_2 \text{ OF } \dots \text{ ARRAY } T_n \text{ OF } T$$

Modula-2 places no limit on the number of dimensions in an array.

If **M** is a variable of type **Matrix**, we can access the component in row **i** and column **j** of **M** by writing either **M[i][j]** or **M[i, j]**. The first form emphasizes the fact that **M** is a one-dimensional array whose components are one-dimensional arrays. The index **i** selects a value of type **ARRAY [1..100] OF INTEGER**; the index **j** selects a component from this array. The other form, **M[i, j]**, is shorter and more closely resembles array access in other programming languages.

Because each row of **M** is an array, rows can appear in assignment statements as whole arrays:

```
M[1] := M[2];
```

Rows of **M** cannot appear in assignment statements with other variables, however, because the type of a row is anonymous. If access to the rows of **M** is important, we can name the type of the rows. The following procedure, which interchanges rows **i** and **j** of a matrix, shows how to do so:

```
TYPE Row = ARRAY [1..100] OF INTEGER;
   Matrix = ARRAY [1..50] OF Row;

PROCEDURE ExchangeRows(VAR M: Matrix; i, j: CARDINAL);
  (* exchanges rows i and j of M *)
  VAR temp: Row;
BEGIN
  temp := M[i];
  M[i] := M[j];
  M[j] := temp;
END ExchangeRows;
```

Because **temp**, **M[i]**, and **M[j]** each has type **Row**, assignments involving any two are legal. This definition of **Matrix** does not affect access to the components of **M**; they can be accessed by writing either **M[i, j]** or **M[i][j]**.

Open Array Parameters

Modula-2 provides two ways of supplying an array parameter to a procedure. In one method, the programmer declares the array type outside the procedure, and then declares the formal parameter and each actual parameter to have this type. Consider the **FindMax** function, which finds the largest component in an array of type **CardArray**:

```
CONST arrayLength = 100;
TYPE CardArray = ARRAY [1..arrayLength] OF CARDINAL;

PROCEDURE FindMax(A: CardArray): CARDINAL;
  (* returns the largest component of A *)
  VAR i, max: CARDINAL;
BEGIN
  max := 0;
  FOR i := 1 TO arrayLength DO
    IF A[i] > max THEN
      max := A[i]
    END
  END;
  RETURN max;
END FindMax;
```

The only problem with **FindMax** is that it accepts only arrays of type **CardArray**, and all such arrays must have exactly 100 components. If we wanted to find the largest component in arrays of lengths 200, 300, and 400 as well, we would need three more procedures, each identical to **FindMax** except for the type of the parameter **A**.

Modula-2 therefore provides a second way to specify the type of an array parameter. Instead of using a type name to specify the type of the formal parameter, the programmer instead declares it to have type

```
ARRAY OF T
```

where **T** is a type name. We call a formal parameter declared in this fashion an *open array parameter*. Note: The programmer may use an **ARRAY OF T** type specification only to declare the type of a formal parameter in a procedure heading, not in ordinary variable or type declarations.

No index type is specified for an open array parameter, allowing the programmer to supply as an actual parameter any array whose components are of type **T**. (As a language extension, TopSpeed Modula-2 also allows the actual parameter to be an expression of type **T**; it is treated as an array with one component.) The index type of an open array parameter is always a subrange of **CARDINAL** whose lower bound is 0. The upper bound varies, depending on the length of the array that the programmer has supplied as an actual parameter. If **A** is an open array parameter, the upper bound of **A**'s index type is **HIGH(A)**. (**HIGH** is a standard function.) Because **A**'s lower index bound is 0, the number of components in the actual parameter is not **HIGH(A)**, but **HIGH(A)** plus 1.

In **FindMax**, if we make **A** an open array parameter, then we can supply as an actual parameter any one-dimensional array whose components have type **CARDINAL**.


```

PROCEDURE FindMax(A: ARRAY OF CARDINAL): CARDINAL;
(* returns the largest component of A *)
VAR i, max: CARDINAL;
BEGIN
  max := 0;
  FOR i := 0 TO HIGH(A) DO
    IF A[i] > max THEN
      max := A[i]
    END
  END;
  RETURN max
END FindMax;

```

When two or more parameters are open arrays, the actual parameters need not have the same length. For example, consider the procedure **InnerProduct**, which computes the inner (or "dot") product of two vectors.

```

PROCEDURE InnerProduct(v, w: ARRAY OF REAL): REAL;
(* returns the inner product of vectors v and w *)
VAR i: CARDINAL;
    sum: REAL;
BEGIN
  IF HIGH(v) # HIGH(w) THEN
    WrStr("Length error in InnerProduct");
    WrLn;
    HALT
  END;
  sum := 0.0;
  FOR i := 0 TO HIGH(v) DO
    sum := sum + v[i] * w[i]
  END;
  RETURN sum
END InnerProduct;

```

The actual parameters in a call of **InnerProduct** may not have the same length, so **InnerProduct** tests whether the lengths match, terminating the program if they do not.

We can use individual components of an open array in the same manner as we use components of ordinary arrays. Modula-2 allows only one operation for whole open arrays, however: The name of an open array may appear as an actual parameter in a procedure call (the corresponding formal parameter must also be an open array parameter). The language does not allow whole array assignment for open array parameters.

String Types

A *string type* is any type of the form

```
ARRAY [0..n] OF CHAR
```


for n greater than or equal to 0. A *string variable* is any variable of a string type. In the following example, **ShortString** and **LongString** are string types; **firstName** and **bookTitle** are string variables:

```
TYPE ShortString = ARRAY [0..9] OF CHAR;
      LongString = ARRAY [0..49] OF CHAR;
VAR  firstName: ShortString;
      bookTitle: LongString;
```

As a language extension, TopSpeed Modula-2 regards any type of the form **ARRAY ... OF CHAR** as a string type—the lower index bound need not be 0.

In previous chapters, we have used the term *string* to mean a sequence of characters enclosed between single or double quotes, such as

```
"Hello, how are you?"
```

For the remainder of this section, we will call this a *string literal*. "Strings" will include both string literals and string variables. The type of a string literal of length n is

```
ARRAY [0..n-1] OF CHAR
```

Thus, the type of **"Hello, how are you?"** is

```
ARRAY [0..18] OF CHAR
```

The quotation marks surrounding a string literal are not part of the string; we do not include them when computing its length.

Strictly speaking, the type of a one-character string literal ("**A**", for example) is **ARRAY [0..0] OF CHAR**, rather than **CHAR**. The distinction is purely academic, however, because a one-character string literal is compatible with type **CHAR**, allowing "**A**" to be treated as being of either type **CHAR** or type **ARRAY [0..0] OF CHAR** as needed.

Because string variables are arrays, Modula-2 allows all the usual array operations for strings:

- selecting a character in a string variable (by indexing)
- assigning one string to another (provided that the strings have the same type)
- supplying a string as a parameter in a procedure call (if the types of actual and formal parameters are the same or if the formal parameter is an open array parameter)
- testing whether two strings of the same type are equal or not equal (allowed in TopSpeed Modula-2 as a language extension)

The only operation Modula-2 defines for string variables but not for arrays in general is the assignment of a string literal to a string variable. This operation has a special property that has important consequences for string manipulation.

When we assign a string literal to a string variable, the length of the literal need not match the length of the variable, provided that the variable is at least as long as the literal. If the literal fails to fill the entire variable, it is copied into the variable starting at the leftmost position and followed by a special terminator, the character 0C. The remaining characters in the variable have undefined values. For example, consider the assignment of the string "Modula-2" to two variables, one of the right length and one too long:

```
VAR short: ARRAY [0..7] OF CHAR;
    long:  ARRAY [0..13] OF CHAR;
    :
    :
short := "Modula-2";    (* the string fills the array *)
long  := "Modula-2";    (* the string fills positions 0 through 7;
                        0C occupies position 8 *)
```

After the two assignments, **short** and **long** look like this:

short:	M	o	d	u	l	a	-	2						
	0	1	2	3	4	5	6	7						

long:	M	o	d	u	l	a	-	2	0C			undefined		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

This special rule does not extend to the assignment of one string variable to another; the types of the two variables must be the same. Thus, the assignment **long := short** is illegal, even though **long** is large enough to hold the contents of **short**, because **short** and **long** do not have the same type.

A Modula-2 string variable, like any array, has a fixed length. However, Modula-2's convention of including a terminating character when a short string literal is assigned to a longer string variable allows the programmer to simulate variable-length strings. If a program needs strings whose lengths will vary as the program runs, but which will never exceed 80 characters, say, the programmer can declare a string type of length 80 and use this type to declare all the variables:

```
TYPE String80 = ARRAY [0..79] OF CHAR;
VAR string1, string2, string3: String80;
```

Whenever a program assigns a string literal to one of these variables, the character 0C automatically terminates the string (unless it fills the variable). When the program copies one of the variables to another, the 0C will also be copied. Thus, the 0C character always marks the end of the variable-length string stored inside the fixed-length array.

The TopSpeed Str Module

Because Modula-2 supplies no operations on strings, TopSpeed Modula-2 includes a library module named **Str** that contains useful procedures for manipulating strings (see the *User's Manual* for details). The programmer may wish to write additional string-handling procedures; Modula-2's open array parameters make it easy to do so.

Record Types

A *record* is a structure whose components, which may be of different types, are selected by names created by the programmer. To declare a record type, a program must list each of the components (*fields*) in the record, giving each a name and a type. For example, suppose we need to represent information about items of merchandise that a store carries. The information for a single item consists of several pieces of data of different types, so it is convenient to collect the information about the item into a record. This information might include an identifying number, a description of the item, the price of the item, and the quantity on hand:

```
TYPE StringType = ARRAY [0..19] OF CHAR;
Item = RECORD
    number: CARDINAL;
    description: StringType;
    price: REAL;
    onHand: INTEGER
END;
```

The fields of a record may be of any type, including array and record types. The **Item** type, for example, contains a field (**description**) of an array type. In a similar fashion, a record may contain another record:

```
TYPE NameString = ARRAY [0..14] OF CHAR;
NameRecord = RECORD
    lastName, firstName: NameString;
    middleInitial: CHAR
END;
PersonnelRecord = RECORD
    number: [0..9999]; (* employee no. *)
    name: NameRecord;
END;
```

We could have avoided declaring the **NameRecord** type, instead declaring the **name** field using the specification of **NameRecord**.

The names of record fields are local to a particular record type; they do not conflict with any other names in a program. In particular, we may use the same identifier to name fields in more than one record type. Thus, the same program may declare both the **Item** and **PersonnelRecord** types, even though they both contain a field named **number**. The same field name cannot appear twice in the same record, however, except within a

nested record. For example, it would be legal to add a field named `number` to `NameRecord`, but not to `PersonnelRecord`.

One advantage of records is the increased flexibility they offer for handling a collection of related data values. Much of the time, a program will access individual fields in a record; at other times, it is more convenient to work with the record as a whole. Modula-2 allows both forms of access.

A reference to a record field consists of a record name and a field name, separated by a period. For example, suppose that we declare the variables `item1` and `item2` as

```
VAR item1, item2: Item;
```

To reference the `number` field in `item1`, we would write

```
item1.number
```

This notation is the same that we use for qualified identifiers. Little chance of confusion arises, however, because the first identifier in a qualified identifier is a module name, not a record name.

One way to change the value of a record variable is to assign values to its fields:

```
item1.number := 143;
item1.description := "Men's black umbrella";
item1.price := 9.99;
item1.onHand := 10;
```

In general, a record field of type `T` has the same properties as a variable of type `T`; it can appear in expressions, on the left side of an assignment, and so on.

As a language extension, TopSpeed Modula-2 provides record aggregates, which closely resemble array aggregates. A record aggregate consists of a record type name followed by a list of constant expressions, which represent the components of the record:

```
item1 := Item(143, "Men's black umbrella", 9.99, 10);
```

Whole records may appear in assignment statements and as parameters in procedure calls. Instead of assigning values piecemeal to the fields of a record, a program may instead change all its fields by whole-record assignment:

```
item1 := item2;
```

This assignment copies `item2.number` to `item1.number`, `item2.description` to `item1.description`, and so on. Assigning one record to another requires that the types of the records be the same. (As with arrays, the terms *same*, *compatible*, and *assignment-compatible* have the same meaning.) Similarly, when we supply a record as a parameter in a procedure call, the record's type must be the same as the type of the corresponding formal parameter. To be the same, two record types must have the same name

or be derived from the same type name (see Chapter 6). It is not enough for the record types to have fields with the same names and types.

As a language extension, TopSpeed Modula-2 also allows a program to compare two records of the same type for equality or inequality.

Example: Updating Inventory Records

Although grouping related variables into a single record is of some value, the most important advantage of record types is their usefulness in creating other types. For example, we can use the `Item` type as the component type for an array; this structure could store information about many items of merchandise. We might also create a pointer type whose values are pointers to records of type `Item`; a program could use this type to dynamically create as many `Item` records as needed.

For example, if we assume that a store stocks no more than 1,000 different items, we could represent its entire inventory by an array of 1,000 records of type `Item`:

```
CONST maxInventory = 1000;
VAR inventory: ARRAY [1..maxInventory] OF Item;
    lastItem: [0..maxInventory];
    (* marks position of last item in inventory array *)
```

One operation that we might perform on such an array is to change the number of items on hand to reflect sales and incoming shipments. The following procedure performs this operation:

```
PROCEDURE UpdateInventory(itemNum: CARDINAL; change: INTEGER;
    VAR found: BOOLEAN);
    (* updates inventory record for itemNum to reflect change in number
       on hand; sets found to TRUE if record is found, FALSE otherwise *)
    VAR currentPos: CARDINAL;
BEGIN
    currentPos := 1;
    LOOP
        IF currentPos > lastItem THEN
            found := FALSE;
            EXIT
        ELSIF itemNum = inventory[currentPos].number THEN
            INC(inventory[currentPos].onHand, change);
            found := TRUE;
            EXIT
        END;
        INC(currentPos)
    END
END UpdateInventory;
```

Designators

The `UpdateInventory` procedure uses expressions such as `inventory[currentPos].number` and `inventory[currentPos].onHand` in

which an index selects an array component and then a field identifier selects a field within the component (because the array components are records). We call such an expression a *designator*. A designator is an identifier (perhaps qualified), possibly followed by any combination of the following:

- a list of index expressions in brackets
- a period, then a field identifier
- the ^ symbol (for dereferencing a pointer)

Modula-2 allows arbitrary nesting of structured types, so designators may become quite complex:

```
inventory[currentPos].description[0]
```

Designators may appear in expressions and on the left sides of assignment statements. In fact, a designator can appear wherever Modula-2 allows a variable, with one exception: The control variable in a **FOR** statement must be an identifier, not a designator.

The WITH Statement

Normally, a programmer refers to a record field by giving a designator for the field, consisting of a designator for the record itself followed by a period and the name of the field. Because designators can be very long, Modula-2 supplies the **WITH** statement. When part of a program repeatedly refers to fields in the same record, the **WITH** statement allows us to access the fields without having to name the record every time.

One common use of the **WITH** statement is to surround a sequence of assignments to the fields of a record:

```
WITH item1 DO
  number := 143;
  description := "Men's black umbrella";
  price := 9.99;
  onHand := 10
END;
```

When it encounters a **WITH** statement, a Modula-2 compiler tries to interpret each identifier mentioned inside the statement as a field in the specified record (*item1* in this case). If the attempt fails, the identifier has its usual meaning. Thus, if a program contains the statement

```
WITH item1 DO
  number := 143;
  itemNum := 59
END;
```


the compiler interprets `number` as `item1.number`, but because `itemNum` is not the name of a field in `item1`, the compiler treats it as though it were not inside the `WITH` statement.

Modula-2 places one important restriction on the `WITH` statement: A program should not try to change the designator from within the body of a `WITH` statement. In general, Modula-2 compilers cannot detect this kind of error, so the programmer must write `WITH` statements carefully. Because of this problem, and because `WITH` statements can make a program difficult to understand, use them sparingly.

Records with Variant Parts

Suppose that we want to write a program to keep track of a computer store's inventory. We decide to use records to store information about each major component of a computer system, such as a system unit, monitor, or printer. However, we want a single record type, because we plan to create an array of such records. The problem is that the information we want to store for a given component depends on what kind of component it is. Each item has a manufacturer, a model, and a price, for example, but other information varies. A record that represents a printer should have a field that can hold the printing speed of the device, but this piece of information is irrelevant for other components.

This kind of problem arises so often that Modula-2 allows a record type to have a *variant part*; different records of that type can have different combinations of fields. The word **CASE** marks the beginning of the variant part:

```

TYPE ComponentType = (systemUnit, monitor, printer);
StringType = ARRAY [0..19] OF CHAR;
MechanismType = (dotMatrix, daisyWheel, laser);
SystemComponent =
  RECORD
    manufacturer: StringType;
    model: StringType;
    CASE kind: ComponentType OF
      systemUnit: memory: CARDINAL;      (* in units of 1K *)
                    numDiskDrives: CARDINAL;
                    hardDisk: BOOLEAN |
      monitor:      color: BOOLEAN;
                    numLines, numColumns: CARDINAL |
      printer:      mechanism: MechanismType;
                    speed: CARDINAL (* characters/second *)
    END;
    price: REAL
  END;
```

Every record of type `SystemComponent` has fields named `manufacturer`, `model`, `kind`, and `price`. The `kind` field is a *tag field*: Its value indicates whether the record represents a system unit, a monitor, or a printer. The value of the tag field selects a particular group of fields, called a *variant*. In the example, if the value of `kind` is `systemUnit`, the record has additional fields named `memory`, `numDiskDrives`, and

hardDisk. The values **monitor** and **printer** select the other two variants. Thus, the record has three possible appearances, depending on the value of the **kind** field.

Because the fields in one variant are never in use at the same time as the fields in the other variants, Modula-2 compilers typically save space by overlaying variants in memory. When the variants require different amounts of space, a compiler allocates enough space for the largest variant.

A program should not reference the fields within the variant part of a record until the program has assigned a value to the tag field. When a program assigns the tag field a value, the variant with a case label corresponding to that value becomes *active*; then the program may legally refer to the fields listed within that variant. Suppose, for example, that we have declared the variable **component** as follows:

```
VAR component: SystemComponent;
```

We can immediately initialize the fields **manufacturer**, **model**, and **price**, which are not in the variant part of the record:

```
component.manufacturer := "IBM";
component.model := "Personal Computer";
component.price := 1000.00;
```

We should not reference the other fields in the record until we have given the tag field a value. We might choose to initialize the tag field to **systemUnit**:

```
component.kind := systemUnit;
```

With this assignment, the **systemUnit** variant becomes active, and we can initialize the fields **memory**, **numDiskDrives**, and **hardDisk**:

```
component.memory := 512;
component.numDiskDrives := 2;
component.hardDisk := FALSE;
```

Notice that we use the same syntax to select all fields of a record, regardless of whether the fields are in the variant part of the record.

A program can change the active variant by assigning a new value to the tag field. Once it has changed the tag field, the program should no longer reference the fields in the old variant. For example, after the assignment

```
component.kind := printer;
```

the **printer** variant becomes active. References to the fields **mechanism** and **speed** are now legal, but references to **memory**, **numDiskDrives**, and **hardDisk** are not.

What happens if a program attempts to reference a field in a variant that is not active? For example, what if the statement


```
WrCard(component.speed, 1);
```

is executed when the `systemUnit` variant is active? In theory, a Modula-2 system could check the value of the `kind` field before each reference to a field in the variant part of the record. Unfortunately, this test must be performed during program execution, not during compilation. Because this test can noticeably increase a program's execution time, TopSpeed Modula-2 does not perform it. Without the check, the call of `WrCard` produces a meaningless number. This example illustrates the danger of using records with variant parts: A reference to an inactive variant generally does not terminate the program, but does result in erroneous operation; unfortunately, this kind of programming error is hardest to fix. To prevent incorrect use of a record with a variant part, a program should always use an `IF` or `CASE` statement to check the value of the tag field if any doubt exists about which variant is active. For example, we can ensure that the call of `WrCard` is valid by using an `IF` statement to check the value of the `kind` field:

```
IF component.kind = printer THEN
  WrCard(component.speed, 1)
ELSE
  WrStr("Illegal reference to speed field");
  WrLn;
  HALT
END;
```

A variant part of a record resembles a `CASE` statement. The `|` symbol separates the cases in a `CASE` statement, as well as the variants in a variant part. (Consequently, programmers tend to format variants in much the same way as cases in a `CASE` statement. In particular, some programmers prefer to put the `|` symbol at the beginning of each variant, rather than at the end. The *User's Manual* adopts this style.) In a `CASE` statement, the value of the case expression indicates which one of several statement sequences will be executed; in a record with a variant part, the value of the tag field indicates which variant is currently active. Furthermore, the syntax of a variant closely resembles the syntax of a case.

Each variant begins with a list of *case labels*: constant expressions of the same type as the tag field. (The tag field is normally subject to the same restrictions as a case expression. That is, the tag field must be of a basic type—other than `REAL` or `LONGREAL`—or an enumeration or subrange type. In TopSpeed Modula-2, the tag field may not be of type `LONGINT` or `LONGCARD`.) No case label can appear in two different variants. The rules for forming a list of case labels are the same as for the `CASE` statement.

After the list of case labels comes a sequence of field lists, which may include both ordinary field lists and variant parts. The field names in this sequence must be different from all other field names in the record, including those belonging to fields outside the variant part, fields in other variants, and the tag field itself.

Modula-2 allows great flexibility in declaring records with variant parts:

- *The tag field of a variant part may be omitted.* The colon that normally follows the name of the tag field must be present even if the programmer omits the name itself; the presence of the colon clearly shows that what remains is the *type* of the tag field. Because omitting the tag field makes it more difficult to check that the program is using variants properly, programmers almost always include the tag field. The only reason for omitting it is to save space. (Modula-2 compilers normally do not allocate memory for a missing tag field. When the programmer omits the tag field, the specification of the tag field type serves only as a form of documentation.)
- *A variant may be empty.* This possibility makes the placement of `|` symbols less critical: at least one `|` must separate two variants, but additional `|` symbols may occur before the first variant, after the last variant, or between variants.
- *A record may have more than one variant part.*
- *A variant part of a record may itself contain a variant part.*
- *A variant part may contain an ELSE clause.* If an **ELSE** clause is present, it must follow the last variant, but precede the **END** that closes the variant part. We can reference the fields listed in the **ELSE** clause only when the current value of the tag field does not match any of the case labels (hence none of the variants are active). An **ELSE** clause can be particularly useful if listing all values of the tag field, either individually or in ranges, would be inconvenient (as might be the case if the tag field were to have type **INTEGER**, for example).

Set Types

In mathematics, a *set* is an unordered collection of elements without duplicates. The elements of a set come from a universe of possible values. For example, if the universe is

{violet, blue, green, yellow, orange, red}

then sets over this universe include

{blue, yellow}
 {red, blue, green}
 {}

({} is the empty set—the set with no elements.) Because sets are unordered, the sets

{blue, yellow}
 {yellow, blue}

are the same. Duplicate elements do not change the value of a set, so the following sets are the same:


```
{orange, red}
{orange, red, orange, red}
{red, orange, red, red}
```

Sets appear in Modula-2 in almost exactly the same form as in mathematics. To declare a set type whose values are sets of colors, we could use the following type declarations:

```
TYPE Colors = (violet, blue, green, yellow, orange, red);
   SetOfColors = SET OF Colors;
```

Colors is the *base type* of **SetOfColors**: the universe from which the elements of a **SetOfColors** value come. A total of 64 (2^6) sets are possible using the six values in **Colors**, so **SetOfColors** has 64 values.

In TopSpeed Modula-2, the base type of a set can be **CARDINAL**, **SHORTCARD**, **CHAR**, an enumeration type, or a subrange of any of these types. Therefore, a TopSpeed Modula-2 set may have as many as 65,536 elements. For example, the following are legal set types in TopSpeed Modula-2:

```
TYPE SetOfChannels = SET OF [2..13];
   YellowRedSet = SET OF [yellow..red];
   CharSet = SET OF CHAR;
```

The following set type is *illegal*:

```
TYPE IntSet = SET OF [-1..10]; (* subrange of INTEGER *)
```

TopSpeed Modula-2's set types greatly extend those in the original language definition, which specifies that the base type of a set must be either an enumeration type (or a subrange of an enumeration type) or a subrange of **CARDINAL**. In the former case, the enumeration type must have at most **N** values, and in the latter case, the upper bound of the subrange must be at most **N - 1**, where **N** is the number of bits in one word or a small multiple thereof.

Modula-2 has one standard set type, called **BITSET**. If it were not a standard type, we could define **BITSET** as follows:

```
TYPE BITSET = SET OF [0..W-1];
```

where **W** is usually the word length of the underlying computer. (In TopSpeed Modula-2, **W** is 16.) The elements of a **BITSET** value normally correspond to the bits in one word of memory. The **BITSET** type is useful primarily for low-level (machine-dependent) programming; we discuss it in more detail in Chapter 10.

Constructing Set Values

Modula-2 allows the programmer to write an expression that constructs a set value from its elements. To construct a set, we specify the name of the set type and the elements to include in the set:

```
SetOfColors{red, blue, green, yellow}
```

We separate the elements of a set with commas, enclose them between the symbols { and }, and precede the entire expression by a (possibly qualified) identifier, which must be the name of a set type.

Including the name of the set type allows a Modula-2 compiler to perform type-checking (notice that a compiler cannot usually deduce the type of a set from its elements). If we omit the name of the set type, the type of the set is **BITSET** by default. If no elements appear between the { and } symbols, the set is empty:

```
SetOfColors{}
```

We can use either individual expressions or ranges to specify set elements. Thus, we could write the set containing **red**, **blue**, **green**, and **yellow** as

```
SetOfColors{red, blue..yellow}
```

Set elements are often constants, but in general may be (nonconstant) expressions. For example, the following set contains one element, the color whose ordinal number is **i**:

```
SetOfColors{VAL(Colors, i)}
```

Set Assignment

A program may assign a set value to a variable of a set type. For example, after the declaration

```
VAR set1, set2, set3: SetOfColors;
```

the following assignment stores the set {**red**, **blue**, **green**, **yellow**} in the variable **set1**:

```
set1 := SetOfColors{red, blue, green, yellow};
```

As usual, the type of the designator on the left side of an assignment must be assignment-compatible with the type of the expression on the right side. For set types, as with arrays and records, the terms *same*, *compatible*, and *assignment-compatible* are synonymous.

Set Operators

The same symbols that represent the four arithmetic operators for **REAL** numbers also represent the four set operators:

+	union
-	difference
*	intersection
/	symmetric difference

If **A** and **B** are sets, then:

The *union* of **A** and **B** is the set of elements belonging to **A** or **B** (or both).

The *difference* of **A** and **B** is the set of elements belonging to **A** but not to **B**.

The *intersection* of **A** and **B** is the set of elements belonging to both **A** and **B**.

The *symmetric difference* of **A** and **B** is the set of elements belonging either to **A** or to **B**, but not both.

After the assignments

```
set1 := SetOfColors{red, blue, yellow};
set2 := SetOfColors{green, red, orange};
```

the following expressions have the indicated values:

Expression	Value
set1 + set2	{red, green, blue, yellow, orange}
set1 - set2	{blue, yellow}
set1 * set2	{red}
set1 / set2	{blue, yellow, green, orange}

The four set operators require that their operands be of the same set type; the result is also of this type.

Set Constants

The existence of set expressions makes possible the declaration of set constants:

```
CONST NoColors = SetOfColors{};
      BlueOnly = SetOfColors{blue};
      RedOnly = SetOfColors{red};
      RedAndBlue = BlueOnly + RedOnly;
```

The declaration of a set constant is no different from the declaration of any other kind of constant. We can use any (constant) set expression to declare a set constant. We must have previously declared a set type (such as **SetOfColors**) that appears in a constant declaration.

Relational Operators

Modula-2 provides five relational operators for sets:

=	equal
<> or #	not equal
<=	subset of
>=	superset of
IN	member of

Each of these produces a **BOOLEAN** result. As usual, the symbols **<>** and **#** are interchangeable. Notice that the relational operators for sets do not include the **<** and **>** symbols.

The **IN** operator tests whether a given value belongs to a set. After the assignment

```
set1 := SetOfColors{blue, green, yellow};
```

the following expressions have the values shown:

<i>Expression</i>	<i>Value</i>
green IN set1	TRUE
orange IN set1	FALSE

In an expression of the form *e* **IN** *s*, the type of *e* must be compatible with the base type of *s*.

The operators **=** and **#** (or **<>**) test whether two sets are equal or not equal. Two sets are equal if they have the same elements. After the assignments

```
set1 := SetOfColors{violet, blue, green, yellow};
set2 := SetOfColors{blue, yellow, green, violet};
set3 := SetOfColors{blue, yellow, violet};
```

the following expressions have the values shown:

<i>Expression</i>	<i>Value</i>
set1 = set2	TRUE
set1 = set3	FALSE
set2 # set3	TRUE

When applied to sets, the operators **<=** and **>=** test the subset and superset relations, respectively. If **A** and **B** are sets, **A** is a *subset* of **B** if every element of **A** is also an element of **B**; **A** is a *superset* of **B** if every element of **B** is an element of **A**. After the assignments

```
set1 := SetOfColors{yellow, orange};
set2 := SetOfColors{green, orange, yellow};
set3 := SetOfColors{yellow, green, orange};
```

the following expressions have the indicated values:

Expression	Value
<code>set1 <= set2</code>	TRUE
<code>set1 >= set2</code>	FALSE
<code>set2 <= set3</code>	TRUE
<code>set2 >= set3</code>	TRUE

The operators `=`, `<>`, `#`, `<=`, and `=>` require operands of the same set type.

Note that `set1 <= set2` has the value **TRUE** if `set1` and `set2` are equal; we say that `set1` is an *improper* subset of `set2` in this case. To test whether `set1` is a *proper* subset of `set2`, the programmer must use the expression

```
(set1 <= set2) & (set1 # set2)
```

Similar remarks apply to the `>=` operator; to test whether `set1` is a proper superset of `set2`, use the expression

```
(set1 >= set2) & (set1 # set2)
```

The INCL and EXCL Procedures

Modula-2 provides two standard procedures, **INCL** ("include") and **EXCL** ("exclude"), for updating the value of a set. **INCL** adds an element to a set (if it is not already there). **EXCL** removes an element from a set; it has no effect if the element does not belong to the set.

```
set1 := SetOfColors(orange, blue);
INCL(set1, green);      (* set1 is now {orange, blue, green} *)
INCL(set1, orange);    (* no change *)
EXCL(set1, blue);      (* set1 is now {orange, green} *)
EXCL(set1, yellow);    (* no change *)
```

If `s` is a variable of type **SetType**, the call

```
INCL(s, x);
```

is equivalent to the statement

```
s := s + SetType(x);
```

and the call

```
EXCL(s, x);
```

is equivalent to

```
s := s - SetType(x);
```

where `x` is any expression belonging to the base type of **SetType**.

INCL and **EXCL** are similar to the standard procedures **INC** and **DEC**. We use the **INCL** and **EXCL** procedures, like **INC** and **DEC**, to make incremental changes to a variable. As with **INC** and **DEC**, a Modula-2 compiler may generate more efficient object code for a call of **INCL** or **EXCL** than for a corresponding assignment statement; furthermore, a call of **INCL** or **EXCL** is often simpler than an assignment statement.

Example: Checking Weekly Schedules

Suppose that a store employs two part-time workers, each of whom works only a few days a week. We can represent each employee's weekly schedule as a value of type **SetOfDays**, where **SetOfDays** is the following set type:

```
TYPE Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
   SetOfDays = SET OF Days;
```

Assume that the store is open seven days a week, and that it needs at least one worker each day, and needs both on Saturday. The following procedure checks the workers' schedules to ensure that at least one person works each day and that both work on Saturday.

```
PROCEDURE CheckSchedule(sched1, sched2: SetOfDays);
  (* checks that every day of the week is included on at
     least one schedule and that Saturday is on both *)
  VAR daysCovered: SetOfDays;
      day: Days;
BEGIN
  daysCovered := sched1 + sched2;
  FOR day := Sun TO Sat DO
    IF NOT (day IN daysCovered) THEN
      WriteDay(day);
      WrStr(" is not covered");
      WrLn
    END
  END;
  IF NOT (Sat IN sched1 * sched2) THEN
    WrStr("Fewer than two workers on Saturday");
    WrLn
  END
END CheckSchedule;
```

CheckSchedule calls a procedure **WriteDay** to write the name of a **Days** value. Although we do not give **WriteDay** here, it is similar to the **WritePlanet** procedure in Chapter 6.

Pointer and Procedure Types

Chapters 6 and 7 covered Modula-2's unstructured and structured types. Pointer and procedure types, the only Modula-2 types not discussed in previous chapters, are the subject of Chapter 8.

The first section explains pointer types and dynamic storage allocation. A pointer value "points to" an anonymous variable created during program execution. Pointers often point to records that themselves contain pointers. This combination of record and pointer types allows the construction of linked data structures, including lists, trees, and graphs.

The second section presents procedure types, whose values are procedures. Procedure types make possible the declaration of procedure variables. A program can assign a procedure to a procedure variable and later activate it by a "call" of the variable. A programmer can also use procedure types to write procedures that accept other procedures as parameters and functions that return procedures.

Pointer types and procedure types have little in common with Modula-2's other types. Pointer values and procedure values are not "data" in the traditional sense. Both are indirect references—to anonymous variables in the case of pointer types, and to procedures in the case of procedure types. A program must "dereference" a pointer value to access the anonymous variable to which it points; a program must "call" a procedure value.

We can think of both pointer values and procedure values as machine-level addresses. A pointer value is the address of an anonymous variable; consequently, we can use pointers as addresses when we do low-level programming (Chapter 10). Similarly, a procedure value is usually the starting address of a procedure's object code.

Pointer Types

Some programs are difficult to write using data structures of fixed sizes. For example, consider a program that maintains a list of names and telephone numbers, allowing the user to add new names at any time and to delete old names. Because the program does not know the maximum number of names in advance, it cannot store the names and numbers in an array. In such a program, *dynamic storage allocation* (and deallocation) is useful. Dynamic storage allocation is the allocation of storage during the execution of a program, in response to requests by the program. The program just described would perform both dynamic allocation and deallocation. When the user wants to add a new name to the list of names and numbers, the program requests enough space to store the name and associated phone number. When the user removes a name from the list, the program can release the space used for that name and phone number and can then reuse it for a new name (or for some other purpose). The space occupied by names and phone numbers will vary during the execution of the program, increasing when the user adds a name to the list and decreasing when the user deletes one.

In Modula-2, variables may be created dynamically as a program executes. Because these variables are not declared in the program, they have no names; they are *anonymous*. A language that allows anonymous variables must provide some way to refer to these variables so that we can assign values to them and later retrieve the values. Modula-2 provides *pointers* for this purpose. A pointer is a reference to an anonymous variable.

Note: In addition to the features described in this section, TopSpeed Modula-2 also provides *based pointers* and *pointer constructors*. Both are language extensions tailored to the architecture of the IBM PC family. For more information about these features, consult the *User's Manual*.

Defining Pointer Types

A pointer variable is not free to point to any anonymous variable; it must point only to anonymous variables of a single type. (Despite not having a name, each anonymous variable has a type.) We must include this type in the specification of the pointer type.

For example, suppose that a program is to dynamically allocate anonymous variables of type **Complex**:

```
TYPE Complex = RECORD
    realPart, imagPart: REAL
END;
```

The following declaration creates a type whose values are pointers to records of type **Complex**:

```
TYPE PtrToComplex = POINTER TO Complex;
```


We can then use **PtrToComplex** to declare pointer variables:

```
VAR ptr1, ptr2: PtrToComplex;
```

The variables **ptr1** and **ptr2** do not take on **Complex** values; instead, the value of each is a pointer to an anonymous variable of type **Complex**. Modula-2 does not allow **ptr1** and **ptr2** to point to ordinary program variables; they can point only to anonymous (dynamically allocated) variables. Furthermore, **ptr1** and **ptr2** are *bound* to a single type (**Complex**, in our example)—they cannot point at anonymous variables of other types. (We can circumvent these restrictions when we use pointers for low-level programming; see Chapter 10.)

The type pointed to can be any standard or programmer-defined type; in practice, it is usually a record type. We can even use an anonymous type to define a pointer type:

```
TYPE PtrToCoordinate = POINTER TO
    RECORD
        x, y: REAL
    END;
```

Creating Anonymous Variables

We can draw the pointer variable **ptr1** as a box containing a pointer to an anonymous variable. Before we have assigned **ptr1** a value, it is undefined:

ptr1 ?

In Modula-2, pointer variables (like variables in general) are not guaranteed any particular initial value. Programs that fail to initialize pointer variables may produce the wrong results or fail during execution.

To create an anonymous variable, a program must call a storage allocation procedure. Modula-2 provides no standard procedure for this purpose. A programmer can either use an allocation procedure imported from a library module (normally provided as part of a Modula-2 system) or write a customized allocation procedure. In the TopSpeed Modula-2 library, the **Storage** module exports an allocation procedure named **ALLOCATE**. **ALLOCATE** requires two parameters: a pointer variable **p** and a **CARDINAL** value **n**. **ALLOCATE** creates an anonymous variable using **n** bytes of storage and modifies **p** to point to this new variable. The programmer need not know the value of **n**; Modula-2 supplies a standard function **SIZE** that will compute it. If **T** is a type name, **SIZE (T)** is the amount of storage required to hold a value of type **T**. (The programmer can also apply **SIZE** to a variable to determine the amount of storage required for the variable.) In our example, **ptr1** is to point to a variable of type **Complex**, so our call to **ALLOCATE** contains **SIZE (Complex)** as the second parameter:


```
ALLOCATE(ptr1, SIZE(Complex));
```

Warning: Be careful to apply **SIZE** to the type of the anonymous variable, not to the pointer type itself:

```
ALLOCATE(ptr1, SIZE(PtrToComplex));    (* wrong! *)
```

Using the name of the pointer type will probably cause the program to allocate the wrong amount of space for the anonymous variable, causing subtle errors during program execution. (Most of the time, the amount of storage required for an anonymous variable is greater than the amount required for a pointer value. When the program allocates too little space for the anonymous variable, it will appear to overlap another anonymous variable; changes to one of the two variables will affect the other in unpredictable ways.) A Modula-2 compiler cannot detect this error, because **ALLOCATE** is a low-level procedure that accepts any **CARDINAL** value as its second parameter.

If a call of **ALLOCATE** fails because of insufficient storage, it terminates the program. For most programs, termination is proper, because the program could not continue without the requested storage. Some programs, however, may wish to recover and continue execution. For this reason, the **TopSpeed Storage** module provides the **Available** procedure:

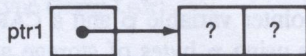
```
PROCEDURE Available(size: CARDINAL): BOOLEAN;
```

By calling **Available**, a program can determine—before calling **ALLOCATE**—whether the desired amount of storage is available:

```
IF Available(SIZE(Complex)) THEN
  ALLOCATE(ptr1, SIZE(Complex))
ELSE
  (* attempt to recover *)
END;
```

Pointer Operations: Dereferencing, Assignment, and Comparison

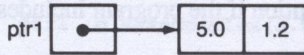
After the call of **ALLOCATE** given earlier, **ptr1** has a value: it points to a newly created (anonymous) variable of type **Complex**:



The new variable has no value, however. The programmer can assign it one in the following way:

```
ptr1^.realPart := 5.0;
ptr1^.imagPart := 1.2;
```


The expression `ptr1^` serves as a name for the record that `ptr1` points to; the programmer can reference fields within this record by using the field names as selectors. After the assignment to the fields of `ptr1^`, it has the following appearance:



The process of forming a name for an anonymous variable by appending the symbol `^` to a pointer variable is called *dereferencing*. Dereferencing is one of the three operations that we can use to form a designator. We can combine it freely with the other two operations: selecting an array component and selecting a record field. If `p` is of type **POINTER TO T**, we can use the designator `p^` in exactly the same way as any variable of type **T** (with one exception: `p^` may not be the control variable in a **FOR** statement).

It is normally an error to dereference a pointer variable whose value is undefined. However, a pointer variable—regardless of its value—may be dereferenced without error if the function **SIZE** is applied to the result. Consider the type **PtrToCoordinate**, whose values are pointers to records of an anonymous type. If `ptr` is a variable of type **PtrToCoordinate**, we must allocate space for an anonymous variable by the call

```
ALLOCATE(ptr, SIZE(ptr^));
```

The type to which `ptr` points has no name, so **SIZE** must have `ptr^` as its parameter (recall that **SIZE** allows either a type name or a variable as its parameter). Because **SIZE** is evaluated during program compilation rather than during program execution, the value of `ptr` is irrelevant.

If a pointer variable points to a record, the dereferenced variable can appear as the designator in a **WITH** statement to allow easy access to the record's fields:

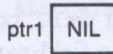
```
WITH ptr1^ DO
  realPart := 5.0;
  imagPart := 1.2
END;
```

As we saw in Chapter 7, the designator in a **WITH** statement should not be changed from within the body of the statement. When a dereferenced pointer is used as the designator, the pointer should not be altered inside the body of the **WITH** statement.

A program can assign the value **NIL** to a pointer variable to indicate that it does not currently point to an anonymous variable:

```
ptr1 := NIL;
```

NIL is a Modula-2 standard identifier that represents the "empty pointer." Modula-2 considers **NIL** to belong to every pointer type, so we may assign it to any pointer variable. After assigning **NIL** to `ptr1`, we draw `ptr1` as follows:

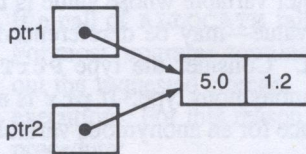


It is illegal to dereference a pointer variable whose value is **NIL**. In TopSpeed Modula-2, this will be checked during program execution if the program includes the compiler directive **(*\$Z+*)**.

We may assign the value of one pointer variable to another pointer variable:

```
ptr2 := ptr1;
```

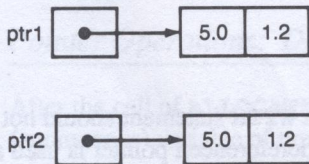
The types of the two variables must be the same; in the case of pointer types, *same*, *compatible*, and *assignment-compatible* have the same meaning. If **ptr1** has a non-NIL value, the assignment makes **ptr2** point to the same anonymous variable as **ptr1**:



The similar-looking assignment

```
ptr2^ := ptr1^;
```

has a completely different effect: it copies the contents of the anonymous variable **ptr1^** to the anonymous variable **ptr2^**. For example, if **ptr1** and **ptr2** point to different anonymous variables before the assignment, then **ptr1** and **ptr2** have the following appearance afterward:



This assignment is legal only if **ptr1** and **ptr2** contain valid pointer values.

Functions may return pointer values:

```

PROCEDURE MakeComplex(realPart, imagPart: REAL): PtrToComplex;
  (* creates an anonymous variable of type Complex and returns
     a pointer to it *)
  VAR temp: PtrToComplex;
BEGIN
  ALLOCATE(temp, SIZE(Complex));
  temp^.realPart := realPart;

```



```

temp^.imagPart := imagPart;
RETURN temp
END MakeComplex;

```

A call of such a function may appear on the right side of an assignment:

```
ptr1 := MakeComplex(5.0, 1.2);
```

The only relational operations allowed on pointer values are tests for equality and inequality. Two pointer values are equal only if both point to the same anonymous variable or if both are *NIL*. Pointers being tested for equality or inequality must have the same type; however, we can compare any pointer for equality or inequality with *NIL*.

Disposing of Anonymous Variables

What happens to an anonymous variable when the variable that points to it is assigned a new value? If another pointer variable does not point to the anonymous variable, it becomes *garbage*—we have no way to access it. Losing anonymous variables this way is not necessarily a problem; however, if a program loses too many, it may eventually exhaust the pool of storage from which it obtains anonymous variables. Fortunately, the same library module that exports a procedure for dynamic storage allocation normally also exports a deallocation procedure. In the **TopSpeed Storage** module, this procedure is called **DEALLOCATE**. Once a program has deallocated an anonymous variable, the space that it occupied can be reused when the program requests a new anonymous variable. The parameters for **DEALLOCATE** are the same as the parameters for **ALLOCATE**. If **ptr1** points to an anonymous variable of type **Complex**, the call

```
DEALLOCATE(ptr1, SIZE(Complex));
```

returns the storage allocated for the anonymous variable to the pool of available storage. The value of **ptr1** after the call is *NIL*.

DEALLOCATE, like **ALLOCATE**, can be dangerous to use. One danger is that the size given when a program deallocates an anonymous variable may not match the size specified when the variable was allocated. Deallocating less than the full size results only in a loss of space, but deallocating too much space may cause subtle errors.

Another danger of using **DEALLOCATE** occurs when two or more pointers point to the same anonymous variable. If both **ptr1** and **ptr2** point to the same variable, then disposing of the variable via **ptr1** leaves **ptr2** pointing to a variable that no longer exists (we say that **ptr2** is a *dangling pointer*). Any subsequent use of **ptr2** may have disastrous effects; worse still, the error can be extremely difficult to find. The following example illustrates the problem:

```

ALLOCATE(ptr1, SIZE(Complex));
ptr1^.realPart := 5.0;
ptr1^.imagPart := 1.2;

```



```

      .
      .
ptr2 := ptr1;
      .
      .
DEALLOCATE(ptr1, SIZE(Complex));
      .
      .
WtReal(ptr2^.realPart, 6, 1);
(* error: ptr2 points to a variable that no longer exists *)

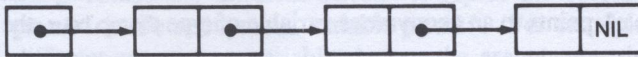
```

The error is easy to spot in this example; however, if these statements were spread over a large program, we would find it difficult to detect—at the call of `WtReal`—that `ptr2` is dangling.

Example: Using Pointers to Construct Linked Lists

Pointers and dynamic storage allocation support the construction of linked structures. A linked structure is a collection of *nodes*, each of which contains one or more pointers to other nodes. A node is just a record in which one or more fields are pointers.

The simplest linked structure is the *linked list*. A linked list consists of a number of nodes, each of which contains one or more pieces of data plus a pointer; the pointers cause the nodes to form a chain:



A pointer variable points to the first node in the list, the first node points to the second node, and so forth. The last node contains `NIL` in its pointer field.

To illustrate linked lists, we show how to define and manipulate a linked list whose nodes contain single characters. First, we must declare two types: `CharNode`, the type of a node in the list, and `CharList`, the type of a pointer to one of these nodes. A value of type `CharNode` will be a record with two fields: `char` (the character stored in the node) and `next` (a pointer to the next node in the list).

```

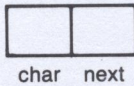
TYPE CharList = POINTER TO CharNode;
CharNode = RECORD
    char: CHAR;
    next: CharList (* points to the next node *)
END;

```

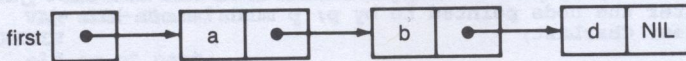
The declaration of `CharList` contains a forward reference to `CharNode`. In general, the declaration of a pointer type may refer to a type not yet declared; without this capability, we could not create linked data structures. The appearance of a forward reference

in a pointer type specification is the only exception to the general Modula-2 rule that a program must declare an entity before using it in another declaration.

We can draw a value of type **CharNode** as a box with two cells named **char** and **next**:



Using this notation, a list of characters would look like this:



The variable **first** (of type **CharList**) points to the first node on the list (of type **CharNode**). The list is empty (has no nodes) if the value of **first** is **NIL**.

We can modify the structure of a linked list in two ways: (1) by inserting a new node into the list and (2) by deleting a node from the list. We now look at each of these operations in turn.

The task of inserting a node can be separated into two cases, depending on whether or not we intend to insert the node after an existing node or at the beginning of the list (so that it becomes the first node).

Suppose first that we need to insert a node containing the character **ch** after an existing node that the variable **p** points to. We first create a new node by allocating space for an anonymous variable of type **CharNode**:

```
ALLOCATE(new, SIZE(CharNode));
```

We then store the desired character in the node:

```
new^.char := ch;
```

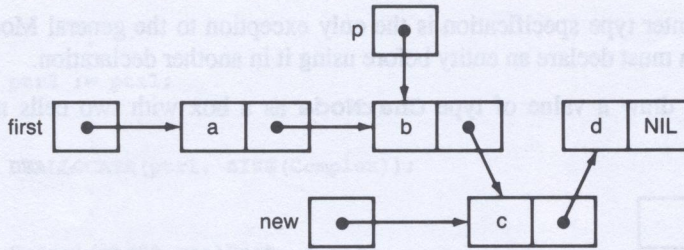
We next set the pointer in the new node to point to the node following **p**:

```
new^.next := p^.next;
```

Finally, we set the pointer in **p** to point to the new node.

```
p^.next := new;
```

The following figure shows the result when we insert a node containing 'c' into a list that contains 'a', 'b', and 'd'. We insert the new node immediately after the node containing 'b'.



The procedure **InsertAfter** performs the operation just described.

```

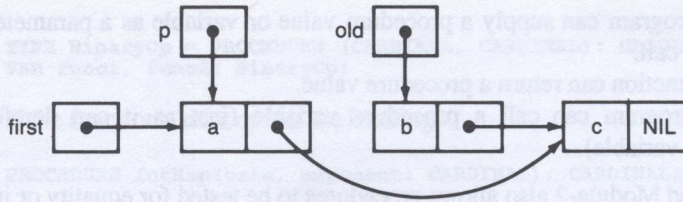
PROCEDURE InsertAfter(p: CharList; ch: CHAR);
  (* creates a node containing ch and adds it to the list just
     after the node pointed to by p; p must be non-NIL *)
  VAR new: CharList;
BEGIN
  ALLOCATE(new, SIZE(CharNode));
  new^.char := ch;
  new^.next := p^.next;
  p^.next := new
END InsertAfter;
  
```

The operation of inserting a node at the beginning of a list is only slightly different. After creating the new node and storing the desired data in it, we make the new node point to the first node on the list, then make **first** point to the new node, as the following procedure shows.

```

PROCEDURE InsertAtFront(VAR first: CharList; ch: CHAR);
  (* creates a node containing ch and adds it to the front of
     the list pointed to by first; first may be NIL *)
  VAR new: CharList;
BEGIN
  ALLOCATE(new, SIZE(CharNode));
  new^.char := ch;
  new^.next := first;  (* make new node point to first node *)
  first := new        (* make first point to new node *)
END InsertAtFront;
  
```

We can also divide into two cases the operation of deleting a node from a list, depending on whether or not another node precedes the node we want to delete. In the first case, suppose that **p** points to the node preceding the one we want to delete. We first set a temporary pointer (stored in the variable **old**) to point to the node to be deleted. We then alter the pointer in **p^** to point to the node following **old^**. Finally, we dispose of the deleted node by calling **DEALLOCATE**. For example, suppose that we delete the second node from a list containing the characters 'a', 'b', and 'c'. The following figure shows the appearance of the list immediately before we destroy **old^**.



The procedure **DeleteAfter** performs the operation just described.

```
PROCEDURE DeleteAfter(p: CharList);
  (* deletes the node immediately following p; p must be non-NIL *)
  VAR old: CharList;
BEGIN
  old := p^.next;
  p^.next := old^.next;
  DEALLOCATE(old, SIZE(CharNode))
END DeleteAfter;
```

The operation of deleting the first node on the list is similar:

```
PROCEDURE DeleteAtFront(VAR first: CharList);
  (* deletes the first node; first must be non-NIL *)
  VAR old: CharList;
BEGIN
  old := first;
  first := first^.next;
  DEALLOCATE(old, SIZE(CharNode))
END DeleteAtFront;
```

Before deleting a node, both **DeleteAfter** and **DeleteAtFront** are careful to save a pointer to the next node in the list. Consider what would happen if **DeleteAtFront** tried to delete the first node without saving a reference to the second node:

```
DEALLOCATE(first, SIZE(CharNode));
first := first^.next;
```

Because **DEALLOCATE** changes the value of **first** (by setting it to **NIL**), the second statement does not have the desired effect.

Procedure Types

Modula-2 allows the creation of *procedure types*, whose values are procedures (or, more precisely, references to procedures). Procedure types have much in common with pointer types. In particular, the operations allowed for procedure values correspond closely to the operations defined for pointers:

A program can assign a procedure value to a procedure variable.

A program can supply a procedure value or variable as a parameter in a procedure call.

A function can return a procedure value.

A program can call a procedure variable (just as it can dereference a pointer variable).

TopSpeed Modula-2 also allows procedures to be tested for equality or inequality.

Defining Procedure Types

The specification of a procedure type resembles a procedure heading; it gives full information about the procedures that belong to the type:

- the number of parameters and the type of each parameter
- whether each parameter is a value parameter or a variable parameter
- the type of value returned by procedures belonging to the type (if they are functions)

We can assign to a variable of this type only procedures that match the specification completely. For example, if we declare **ProcType** as

```
TYPE ProcType = PROCEDURE (REAL, VAR INTEGER);
```

then a variable of type **ProcType** can store only procedures with two parameters, where the first is a value parameter of type **REAL** and the second a variable parameter of type **INTEGER**. If we declare **FuncType** as

```
TYPE FuncType = PROCEDURE (CARDINAL): CARDINAL;
```

then a variable of type **FuncType** can store any function that has one value parameter of type **CARDINAL** and returns a result of type **CARDINAL**. (TopSpeed Modula-2 requires only assignment compatibility for value parameters. For example, a variable of type **FuncType** could be assigned a function whose parameter has type **INTEGER**.)

Modula-2 provides one standard procedure type, **PROC**, which has the following definition:

```
TYPE PROC = PROCEDURE;
```

Any proper procedure that has no parameters is of type **PROC**.

Procedure Assignment

We can give a value to a procedure variable by assignment. The right side of the assignment may be either a procedure name, a procedure variable, or a call of a function that returns a procedure value. For example, suppose that **func1** and **func2** are variables of type **BinaryOp**:


```
TYPE BinaryOp = PROCEDURE (CARDINAL, CARDINAL): CARDINAL;
VAR func1, func2: BinaryOp;
```

If the function **IntExp** has the heading

```
PROCEDURE IntExp(base, exponent: CARDINAL): CARDINAL;
```

we can assign it to **func1**:

```
func1 := IntExp;
```

(The absence of parentheses following the name **IntExp** signals the compiler that **IntExp** itself—not the value returned by a call of **IntExp**—is to be assigned to **func1**.) We can also copy the procedure reference stored in **func1** to **func2**:

```
func2 := func1;
```

When the name of a procedure appears on the right side of an assignment to a procedure variable, the heading of the procedure must match the specification of the variable's type, as described earlier. When we assign a procedure variable to another procedure variable, the types of the two variables must be the same, in the usual Modula-2 sense. (For procedure types, the terms *same*, *compatible*, and *assignment-compatible* are synonymous.)

As a language extension, TopSpeed Modula-2 provides the value **NULLPROC**, which can be assigned to any procedure variable. **NULLPROC** serves the same role for procedure variables that **NIL** serves for pointer variables—it represents an "empty reference."

Modula-2 forbids use of the following kinds of procedures as procedure values:

- standard procedures (such as **ABS** and **INC**)
- procedures that are local to other procedures

These procedures cannot be assigned to procedure variables, used as parameters, or returned by functions; Modula-2 compilers will detect violations of this rule. Note that procedures exported by library modules do not fall into either of these categories.

Calling a Procedure Variable

We can call a procedure variable, once it has a value, in the same way as an ordinary procedure (or function). Thus, after the assignments to **func1** and **func2**, the following statements are equivalent:

```
WrCard(IntExp(1, j), 1);
WrCard(func1(1, j), 1);
WrCard(func2(1, j), 1);
```


The first statement calls `IntExp` directly. The second calls `IntExp` indirectly through the variable `func1`, which contains a reference to `IntExp`. And, because the value of `func1` was copied to `func2`, the call of `func2` in the third statement is also a call of `IntExp`. A procedure variable has no particular initial value; attempting to call a procedure variable before we have assigned it a value will have unpredictable results. Attempting to call a variable that has been assigned `NULLPROC` causes a run-time error.

Procedures Used as Parameters

Procedure types make it possible to supply a procedure as a parameter to a call of another procedure. For example, the following function, when we give it a function `f` as a parameter, finds the smallest nonnegative integer `n` for which `f(n)` is 0. The parameter `f` must be a function that has a single value parameter of type `INTEGER` and produces an `INTEGER` result.

```
TYPE IntToInt = PROCEDURE (INTEGER): INTEGER;

PROCEDURE FindZero(f: IntToInt): INTEGER;
  (* returns the smallest nonnegative integer n for which
     f(n) = 0; does not terminate if f(n) is never 0 *)
  VAR n: INTEGER;
  BEGIN
    n := 0;
    WHILE f(n) # 0 DO
      INC(n)
    END;
    RETURN n
  END FindZero;
```

If `Poly` is the following function:

```
PROCEDURE Poly(i: INTEGER): INTEGER;
  BEGIN
    RETURN i * i + i - 12
  END Poly;
```

then the call

```
WrCard(FindZero(Poly), 1);
```

writes the number 3, because `Poly(0)` is -12, `Poly(1)` is -10, `Poly(2)` is -6, and `Poly(3)` is 0.

Functions That Return Procedure Values

Modula-2 functions may return values of a procedure type. The following function, when given one of the characters `'+'`, `'-'`, `'*'`, or `'/'`, returns a function to perform the corresponding arithmetic operation.


```

TYPE ArithmeticOp = PROCEDURE (REAL, REAL): REAL;

PROCEDURE FindOp(operator: CHAR): ArithmeticOp;
(* returns a function corresponding to a one-character
operator; character must one of +, -, *, or / *)
BEGIN
  CASE operator OF
    '+': RETURN Add      |
    '-': RETURN Subtract |
    '*': RETURN Multiply |
    '/': RETURN Divide
  END
END FindOp;

```

Add, Subtract, Multiply, and Divide are programmer-defined functions of type **ArithmeticOp**.

A Module-2 program consists of a program module (the main program) and any number of library modules from which the program module imports entities (constants, types, variables, and procedures). These library modules may in turn import from other library modules, which are also considered part of the program. Each Module-2 system provides a basic collection of library modules; the programmer is free to write additional library modules to use in a single program or in many programs. Program modules and library modules may contain smaller, nested modules hidden from the rest of the program; these are called *local modules*.

Modules serve as convenient repositories of related entities. Not all the entities that belong to a module are available for use by other modules; however, Modules act as visible entities declared inside a module are not visible outside the module unless the module exports them.

Writing a program as a collection of modules is neither simpler nor a single unit—offers many advantages:

- **Modules are a powerful tool for creating large programs.** A programmer can design a large program as a collection of modules connected through clearly defined interfaces. Writing and testing the program is simpler because the programmer can work on one module at a time, using the services provided by other modules but ignoring the details of how those services work (the principle of abstraction). Module design is particularly useful when a team of programmers develops a program. Each module can be assigned to a single member of the team, who writes and tests the module knowing only the interfaces of that module and others in the program.

Chapter 9

Modules

A Modula-2 program consists of a program module (the main program) and any number of *library modules* from which the program module imports entities (constants, types, variables, and procedures). These library modules may in turn import from other library modules, which are also considered part of the program. Each Modula-2 system provides a basic collection of library modules; the programmer is free to write additional library modules to use in a single program or in many programs. Program modules and library modules may contain smaller, nested modules hidden from the rest of the program; these are called *local modules*.

Modules serve as convenient repositories of related entities. Not all the entities that belong to a module are available for use by other modules, however. Modules act as walls: entities declared inside a module are not visible outside the module unless the module *exports* them.

Writing a program as a collection of modules—rather than as a single unit—offers many advantages:

- **Modules are a powerful tool for creating large programs.** A programmer can design a large program as a collection of modules connected through clearly defined interfaces. Writing and testing the program is simpler because the programmer can work on one module at a time, using the services provided by other modules but ignoring the details of how those modules work (the principle of *abstraction*). Modular design is particularly useful when a team of programmers develops a program. Each module can be assigned to a single member of the team, who writes and tests the module knowing only the specifications of that module and others in the program.

- *Modules help make programs easier to modify.* Modules make *information hiding* possible: a programmer can change details that are hidden inside a module without affecting the rest of the program, provided that the module's specifications remain the same. For example, when a module exports a procedure, the programmer can modify the body of the procedure to correct an error or to take advantage of a better algorithm. Similarly, the programmer can hide the representation of a data structure in a module and change it without affecting other modules in the program.
- *Modules help make programs more portable.* A programmer can hide machine-dependent portions of a program in a module (or a few modules). When transporting the program to another computer, the programmer need only change this module to account for machine differences.
- *Modules make separate compilation possible.* Modules provide a natural way of dividing a program into pieces that a compiler can process separately. A change to one module requires recompilation of only that module, not of the entire program, provided that the module's interface with the rest of the program did not change. (The programmer may need to relink the modules, however.) In Modula-2, separate compilation involves no loss of safety, because compilers perform full type-checking across module boundaries.
- *Modules enable programmers to develop libraries of reusable code.* If a library module developed for one program is written in a general fashion, programmers can reuse the module in other programs. Reusable modules not only save programming time, but also contribute to program reliability because they have undergone testing in previous programs.

Although the programs in Chapters 1–8 used library modules, these chapters did not discuss how to write and compile library modules, nor did the programs in those chapters use local modules. Library modules and local modules are the subject of this chapter.

Library Modules

A library module has two parts: the *definition part* and the *implementation part*. The definition part contains only definitions of entities (constants, types, variables, and procedures) that the module exports; it defines the interface of the library module with the rest of the program. The implementation part describes how to implement the services listed in the definition part. To emphasize that the definition and implementation parts of a library module are themselves complete program units, the first is often called a *definition module* and the second an *implementation module*. TopSpeed Modula-2—like most Modula-2 systems—requires that definition and implementation modules be kept in separate files.

A definition module contains *definitions*, not declarations. Definitions resemble declarations, but in some cases provide less information. The definition of a procedure consists

of the procedure heading only, not the body of the procedure. The definition of a type may omit the specification of the type, giving only the name of the type.

The parameters of the trigonometric functions in the TopSpeed **MATHLIB** module must be expressed in radians, not in degrees. Programs that measure angles in degrees need functions to convert from degrees to radians and vice-versa (call these functions **DegreesToRadians** and **RadiansToDegrees**). Because these functions are useful in more than one program, it makes sense to create a library module called **AngleConversions** that exports the two functions. Once written and tested, the module becomes a reusable tool that any number of programs can use. The definition part of **AngleConversions** might look as follows:

```
DEFINITION MODULE AngleConversions;
  (* exports procedures for converting between degrees and radians *)

  PROCEDURE DegreesToRadians(degrees: LONGREAL): LONGREAL;
    (* converts angle in degrees to radians *)

  PROCEDURE RadiansToDegrees(radians: LONGREAL): LONGREAL;
    (* converts angle in radians to degrees *)

END AngleConversions.
```

The definition part of a library module specifies what services the module provides. The definition part of **AngleConversions** gives only the headings of **DegreesToRadians** and **RadiansToDegrees**; a comment accompanying each function describes what it does. The implementation part hides the internal details of the module (here, the bodies of the functions).

A module that imports an entity exported by a library module is said to be a *client* of the library module. The following module, **ComputeThirdSide**, is a client of the library module **AngleConversions**, from which it imports the function **DegreesToRadians** (**ComputeThirdSide** is also a client of the library modules **IO** and **MATHLIB**):

```
MODULE ComputeThirdSide;
  (* computes the length of the third side of a triangle from the
    lengths of the other two sides and the angle between the sides *)

  FROM IO IMPORT WrStr, WrLn, RdLngReal, WrLngReal;
  FROM MATHLIB IMPORT Sqrt, Cos;
  FROM AngleConversions IMPORT DegreesToRadians;

  VAR sidel, side2, side3, angle: LONGREAL;

BEGIN
  WrStr("Enter length of one side: ");
  sidel := RdLngReal();
  WrStr("Enter length of other side: ");
  side2 := RdLngReal();
  WrStr("Enter angle between sides (in degrees): ");
  angle := RdLngReal();
```



```

side3 := Sqrt(side1 * side1 + side2 * side2 -
              2.0 * side1 * side2 * Cos(DegreesToRadians(angle)));
(* law of cosines *)

WrStr("The length of the third side is: ");
WrLngReal(side3, 15, 1);
WrLn
END ComputeThirdSide.

```

Notice that **ComputeThirdSide** knows nothing of how **DegreesToRadians** works; in fact, we can compile **ComputeThirdSide** before we write the implementation part of **AngleConversions**.

The implementation part of a library module "fills in the blanks" left by the definition part—it supplies all information missing from the definition part. Procedures whose headings appear in the definition part must have complete declarations in the implementation part. If we have omitted the specification of a type in the definition part, we must declare it in full in the implementation part.

The implementation part of a library module begins with the words **IMPLEMENTATION MODULE**, followed by the module name (which must match the name given in the definition part of the module). Here is the implementation part of **AngleConversions**:

```

IMPLEMENTATION MODULE AngleConversions;

CONST twoPi = 2.0 * 3.14159265;

PROCEDURE DegreesToRadians(degrees: LONGREAL): LONGREAL;
(* converts angle in degrees to radians *)
BEGIN
  RETURN (twoPi * degrees) / 360.0
END DegreesToRadians;

PROCEDURE RadiansToDegrees(radians: LONGREAL): LONGREAL;
(* converts angle in radians to degrees *)
BEGIN
  RETURN (360.0 * radians) / twoPi
END RadiansToDegrees;

END AngleConversions.

```

In addition to supplying information missing in the definition part of the module (in the case of **AngleConversions**, complete declarations of the procedures **DegreesToRadians** and **RadiansToDegrees**), the implementation part may declare entities not mentioned in the definition part. These entities (such as the constant **twoPi** in **AngleConversions**) are local to the implementation module: they are not visible elsewhere in the program. The implementation part of a module may have a body (the word **BEGIN**, followed by a sequence of statements), which typically serves to initialize the module's variables. (**AngleConversions** has no variables, so its body is empty. When the body of a module is empty, we may omit the word **BEGIN**.)

A Modula-2 compiler uses the information in the definition part of **AngleConversions** to check for proper use of the procedures it exports (for example, that they are

called with parameters of correct number and appropriate types). When we later compile the implementation part of **AngleConversions**, the compiler will check that it is consistent with the definition part (for example, that procedure headings match). In this way, a Modula-2 system can perform type-checking across module boundaries.

Definition Modules

A definition module contains *definitions* of constants, types, variables, and procedures. Definitions are similar to declarations, but may provide less information.

Constants

Constant definitions are identical to normal constant declarations:

```
CONST MaxOpenFiles = 15;
```

Types

We can define types in two ways. One possibility is to use a normal type declaration:

```
TYPE Complex = RECORD
    realPart, imagPart: REAL
END;
```

A type whose full declaration appears in a definition module is subject to *transparent export*; the structure of the type is visible to client modules. Transparent export of a record type causes the names of the record's fields to be exported; a module that imports **Complex** automatically imports the field names **realPart** and **imagPart**. Similarly, the transparent export of an enumeration type causes the export of the names of the enumeration constants as well. For example, if a definition module contains the definition

```
TYPE Scale = (do, re, mi, fa, so, la, ti);
```

then the module exports the names **do**, **re**, **mi**, **fa**, **so**, **la**, and **ti** along with **Scale**.

Alternatively, we can omit the equals sign and type specification in a type definition:

```
TYPE Complex;
```

We call a type defined in this fashion an *opaque type*, because client modules do not know its structure. A full declaration of an opaque type must appear in the matching implementation module. We will see later that a value of an opaque type must be a pointer; thus, a Modula-2 compiler always knows how much space to allocate for a variable of an opaque type.

Variables

Variable definitions are identical to ordinary variable declarations:

```
VAR OK: BOOLEAN;
```

Procedures

The definition of a procedure consists of just the heading of the procedure:

```
PROCEDURE Add(c1, c2: Complex): Complex;
```

A full declaration of the procedure (including both the heading and body) must appear in the matching implementation module.

A definition module may include import lists. Import lists are not mandatory, and in fact many definition modules do not contain any. Because the sole purpose of the definition part of a library module is to supply definitions of the entities that the module exports, the only entities it might need to import are constants and types, which can be used to define other entities. A definition module might import a constant that it needs in order to declare an array type, or it might import a type to declare another type or to specify the type of a procedure parameter. For example, the following definition module, which exports procedures for file manipulation, imports a file type to use in declaring the procedures' parameters:

```
DEFINITION MODULE FileOperations;
  (* exports procedures for performing file operations *)

  FROM FIO IMPORT File;

  PROCEDURE Copy(VAR f1, f2: File);
    (* copies f1 to f2 *)

  PROCEDURE Compare(VAR f1, f2: File): BOOLEAN;
    (* returns TRUE if the contents of f1 and f2 are the same,
       FALSE otherwise *)

END FileOperations.
```

Example: The Definition Part of a Stack Module

Suppose that we are writing a program called **Reverse** that will read a character string and print its reversal. **Reverse** is easy to write if we use a stack to store the string. As **Reverse** reads characters, it pushes them onto the stack, stopping when it reaches the end of the string. **Reverse** then pops characters from the stack until it is empty, writing each character as it is popped.

We could either declare the stack within **Reverse** or place it in a module of its own (called **CharStackModule**, say). The latter approach offers several advantages:

- **Reverse** becomes simpler and easier to understand, because it contains only natural stack operations such as **Push** and **Pop**.
- **Reverse** cannot affect correct functioning of the stack, because we can hide the variables that make up the stack within **CharStackModule**, limiting access to these variables to the procedures exported by **CharStackModule**. **Reverse** cannot corrupt the stack, say by accidentally changing the value of a variable that marks the top of the stack.
- We can change the implementation of the stack later without changing **Reverse**. For example, if we use an array to represent the stack, stack overflow may occur too often. If so, we might switch to a dynamic data structure (a linked list, for example) to implement the stack.
- Other programs could use **CharStackModule**.

Our informal description of **Reverse** reveals that we need three stack operations: (1) push a character onto the stack, (2) pop a character off the stack, and (3) test whether the stack is empty. (The implementation part of **CharStackModule** will handle stack initialization.) Therefore, **CharStackModule** exports the three procedures **Push**, **Pop**, and **IsEmpty**:

```

DEFINITION MODULE CharStackModule;
  (* exports procedures for manipulating a hidden stack of characters *)

  PROCEDURE Push(ch: CHAR);
    (* pushes ch onto the stack; terminates program if stack is full *)

  PROCEDURE Pop(VAR ch: CHAR);
    (* stores the top element of the stack into ch, then pops
      the stack; terminates program if stack is empty *)

  PROCEDURE IsEmpty(): BOOLEAN;
    (* returns TRUE if the stack is empty, FALSE otherwise *)

END CharStackModule.

```

The definition part of **CharStackModule** furnishes all the information that **Reverse** needs to know about **Push**, **Pop**, and **IsEmpty**. **Reverse** does not need to know the details of how **Push**, **Pop**, and **IsEmpty** work, nor does it need to know the representation of the stack. As a matter of style, Modula-2 programmers customarily accompany each definition in a definition module with a comment explaining the entity being defined. These comments must be complete and accurate, particularly when other programmers must use the module.

We can write **Reverse** using our knowledge of the definition part of **CharStackModule**; we need not worry about the implementation part of the module (which appears later in this section). **Reverse** imports **Push**, **Pop**, and **IsEmpty** from **CharStackModule** in the same way that it imports from **IO**.


```

MODULE Reverse;
  (* reverses a string typed by the user *)

  FROM IO IMPORT RdKey, WrChar, WrStr, WrLn;
  FROM CharStackModule IMPORT Push, Pop, IsEmpty;

  CONST cr = 15C;      (* carriage return *)

  VAR ch: CHAR;

BEGIN
  WrStr("Enter string: ");
  ch := RdKey();
  WHILE ch # cr DO
    WrChar(ch);      (* echo character read *)
    Push(ch);
    ch := RdKey();
  END;
  WrLn;

  WrStr("The reversal is: ");
  WHILE NOT IsEmpty() DO
    Pop(ch);
    WrChar(ch);
  END;
  WrLn;
END Reverse.

```

Implementation Modules

An implementation module has exactly the same syntax as a program module, except that the former begins with the words **IMPLEMENTATION MODULE** instead of **MODULE**.

Implementation modules, like program modules, may contain both import lists and declarations. Import lists allow an implementation module to import entities from other library modules. Declarations serve two purposes: (1) to fill in details missing from the matching definition module and (2) to declare entities that are strictly local to the implementation module.

Any entities *defined* in the definition part of a library module are visible in the implementation part as well. For example, the following definition module defines two entities, the type **NameString** and the array **charNames**:

```

DEFINITION MODULE ControlChars;
  (* exports an array containing names of ASCII control characters *)

  TYPE NameString = ARRAY [0..2] OF CHAR;

  VAR charNames: ARRAY [0C..37C] OF NameString;

END ControlChars.

```

Both **NameString** and **charNames** are visible in the corresponding implementation module (although it does not use **NameString**):


```

IMPLEMENTATION MODULE ControlChars;
BEGIN
  charNames[0C] := "nul";
  charNames[1C] := "soh";
  .
  .
  charNames[36C] := "rs";
  charNames[37C] := "us"
END ControlChars.

```

In TopSpeed Modula-2, entities *imported* into the definition part of a module are automatically visible in the implementation part. In many other Modula-2 systems, however, if both the definition and implementation parts need the same entity, both must import it. For compatibility with these systems, TopSpeed Modula-2 allows the implementation part to import entities already imported in the definition part. For example, the implementation part of **FileOperations** can import the **File** type, even though the definition part (shown earlier in this section) imports it:

```

IMPLEMENTATION MODULE FileOperations;
  FROM FIO IMPORT File;
  (* allowed by TopSpeed Modula-2, but not required *)
  .
  .
END FileOperations.

```

An implementation module may contain declarations of all five kinds of Modula-2 entities:

- **Constants.** Constants defined in the definition part of a module are automatically visible in the implementation part. Any constants declared in an implementation module are local to the implementation module; they are not visible outside the implementation module.
- **Types.** Types defined in the definition part of a module are automatically visible in the implementation part. If the specification of the type appears in the definition part, a declaration of the type must *not* appear in the implementation part. However, if the definition part omits the type specification, a full declaration of the type must appear in the implementation part. The implementation part may also declare new types not mentioned in the definition part; these are local to the implementation module.
- **Variables.** Variables defined in the definition part of a module may not be declared in the implementation part, because they are automatically visible there. All variables declared in an implementation module are local to the module.
- **Procedures.** The implementation part of a module must contain a full declaration of every procedure whose heading appears in the definition part. (The full declaration must agree in every detail with the heading given in the definition part; in particular, parameters must have the same names and types and appear in the same order.) In

addition, an implementation module may declare other procedures that are local to the module.

- *Modules.* An implementation module may have modules declared inside it. These *local modules* are discussed later in the chapter.

All variables declared at the outermost level of a library module (that is, not nested inside a procedure), whether declared in the definition part or the implementation part, are *static*: they retain their values throughout the lifetime of the program. (In contrast, variables declared in a procedure lose their values whenever the procedure returns.) Consequently, we need initialize the variables belonging to a library module only once—when the program begins execution. So that a library module can initialize its variables, *Module-2* allows the implementation part of the module to have a body; the statements in the body are executed at the beginning of program execution, before the body of the program module is executed.

Example: The Implementation Part of a Stack Module

Before we write the implementation part of **CharStackModule**, we must decide how it will represent a stack of characters. We have many choices; for simplicity, we use a one-dimensional array of characters called **stackArray** together with a variable **topOfStack** indicating the top of the stack. In addition to declarations of the procedures listed in the definition part of **CharStackModule**, the implementation part will contain declarations of **stackArray**, **topOfStack**, and a constant (**stackSize**) used to specify the stack's maximum size. These entities are strictly local to the implementation part of **CharStackModule**.

Before a client module can call the procedures **Push**, **Pop**, and **IsEmpty**, **CharStackModule** must initialize the stack by assigning the value 0 to **topOfStack**; the body of the module performs this assignment.

```
IMPLEMENTATION MODULE CharStackModule;
```

```
  FROM IO IMPORT WrStr, WrLn;
```

```
  CONST stackSize = 100; (* maximum size of stack *)
```

```
  VAR stackArray: ARRAY [1..stackSize] OF CHAR;
      topOfStack: [0..stackSize+1];
```

```
  PROCEDURE Push(ch: CHAR);
```

```
    (* pushes ch onto the stack; terminates program if stack is full *)
```

```
  BEGIN
```

```
    INC(topOfStack);
```

```
    IF topOfStack > stackSize THEN
```

```
      WrStr("Stack overflow in CharStackModule.Push");
```

```
      WrLn;
```

```
      HALT
```

```
    END;
```

```
    stackArray[topOfStack] := ch
```

```
  END Push;
```



```

PROCEDURE Pop(VAR ch: CHAR);
  (* stores the top element of the stack into ch, then pops
   the stack; terminates program if stack is empty *)
BEGIN
  IF topOfStack = 0 THEN
    WrStr("Stack underflow in CharStackModule.Pop");
    WrLn;
    HALT
  END;
  ch := stackArray[topOfStack];
  DEC(topOfStack)
END Pop;

PROCEDURE IsEmpty(): BOOLEAN;
  (* returns TRUE if the stack is empty, FALSE otherwise *)
BEGIN
  RETURN topOfStack = 0
END IsEmpty;

BEGIN
  topOfStack := 0
END CharStackModule.

```

The body of **CharStackModule** will be executed before the body of **Reverse**, ensuring proper initialization of the stack before **Reverse** performs any operations on it.

CharStackModule reacts to stack overflow or underflow by writing an error message and terminating the program. A better version of **CharStackModule** would signal the error to the client module, allowing it to decide whether to terminate the program or continue execution.

Order of Module Initialization

Because a program may contain any number of library modules, each of which may have a body, we need to know the order of module initialization. The rule is simple: The bodies of all imported modules must be executed before the body of a client module can be executed. (This order ensures proper initialization of each library module before a client performs any operations on it.) When the client module imports from several modules, they are initialized in the order they are imported. The program module is always executed last, after the bodies of all imported modules have been executed.

For example, suppose that a program module **M1** imports from a library module **M2**, whose implementation part imports from another library module **M3**. **M1** is a client of **M2**, so **M2** must be initialized before the body of **M1** can be executed. But **M2** is a client of **M3**, so **M3** must be initialized before **M2** can execute. Thus, the order of initialization will be **M3** first, **M2** second, and **M1** last.

A more interesting case occurs when **M1** imports from both **M2** and **M3**, which in turn import from **M4**. The order of initialization depends on the order of the import lists in **M1**. Assuming that the import list for **M2** precedes the list for **M3**, the order of initialization will be **M4**, **M2**, **M3**, then **M1**.

The order of initialization is undefined if the pattern of importation among two or more implementation modules forms a circle. In this case, the linker chooses the order in which the modules are initialized. If this choice does not correspond to what the programmer had in mind, the program will not function properly.

The programmer should try to avoid creating such loops. If a loop is unavoidable, the programmer must ensure that the initialization of each module in the loop does not depend on the initialization of any other module in the loop.

Opaque Types

A definition module can export a type whose structure is unknown, by defining just the name of the type. For example, the definition part of a library module called **ComplexNumbers** might contain the following definition:

```
TYPE Complex;
```

A type declared in this fashion is subject to *opaque export*, because client modules have no knowledge of the type's structure. We call the type itself an *opaque type*.

A client module that imports an opaque type may use it in the same fashion as any Modula-2 type; in particular, the module may use the type to declare variables. Because client modules do not know the structure of the type, the only operations they may perform on variables of the type are assignment and tests for equality and inequality. The module that exports the opaque type must supply procedures to perform all other operations that clients may need. Because these procedures are declared in the same implementation module as the opaque type, they have access to the structure of the type.

When the definition part of a library module defines an opaque type, the implementation part of the module must include a full declaration of the type. The full declaration of an opaque type must specify it to be a pointer type. Thus, the implementation part of the **ComplexNumbers** module might contain the following declaration:

```
TYPE Complex = POINTER TO RECORD
    realPart, imagPart: REAL
END;
```

The restriction of opaque types to pointers is a consequence of the separation of the definition and implementation parts of a library module. Suppose that a program module **P** imports an opaque type **T** from a library module **M**. When we compile **P**, the compiler must know how much storage to allocate for variables of type **T**. We have hidden the specification of **T** in the implementation part of **M**, however, which is not accessible during the compilation of **P**. The requirement that **T** be a pointer type solves the problem, because all pointer types require the same amount of storage.

Requiring values of an opaque type to be pointers is not nearly as harsh as it may seem. In fact, it is hardly a restriction at all, because we can define an opaque type as a pointer to a record type with as many fields as we want. Many opaque types represent linked structures (stacks, queues, or trees, for example) and therefore would be pointer types in any event. Using a pointer type as an opaque type can even be advantageous. Because values of the opaque type are pointers to anonymous variables, we can export procedures to create and destroy these variables, allowing better management of space.

Client modules may not take advantage of the fact that values of an opaque type are pointers. Writing a client module that tries to use a variable of an opaque type as though it were a pointer (by dereferencing it, for example) is one of the most common mistakes that beginning Modula-2 programmers make.

Example: A Module That Exports an Opaque Stack Type

The module **CharStackModule** is fine for a program that needs only one stack of characters, but is useless if we need two or more stacks. We can increase the value of this module by having it export a **CharStack** type and a collection of procedures that operate on **CharStack** variables. A client module can then import **CharStack** and use it to declare any number of stacks.

Starting with **CharStackModule**, we will create a new module called **CharacterStacks** that exports type **CharStack** and procedures **Push**, **Pop**, and **IsEmpty**. Each procedure must now have an additional parameter of type **CharStack**, so that a call of the procedure can specify which variable of type **CharStack** to use for the operation. As a first attempt at writing the definition part of **CharacterStacks**, we will declare **CharStack** to be a record containing a stack array and a top-of-stack index.

```

DEFINITION MODULE CharacterStacks;
  (* exports type CharStack (stack of characters) and
    procedures for manipulating variables of type CharStack *)

  CONST stackSize = 100;    (* maximum size of a CharStack *)

  TYPE CharStack = RECORD
    stackArray: ARRAY [1..stackSize] OF CHAR;
    topOfStack: [0..stackSize+1]
  END;

  (* transparent export of CharStack *)

  PROCEDURE Push(VAR s: CharStack; ch: CHAR);
    (* pushes ch onto stack s; terminates program if stack is full *)

  PROCEDURE Pop(VAR s: CharStack; VAR ch: CHAR);
    (* stores top element of stack s into ch, then pops s;
      terminates program if stack is empty *)

  PROCEDURE IsEmpty(s: CharStack): BOOLEAN;
    (* returns TRUE if s is empty, FALSE otherwise *)

END CharacterStacks.
```


This module is perfectly legal in Modula-2, but leaves much to be desired. If we decide to increase the maximum stack size, we must recompile all modules that import from **CharacterStacks**. We must also recompile if the representation of **CharStack** ever changes. Furthermore, by revealing the representation of a character stack, the definition part of **CharacterStacks** allows client modules to directly access the components of the stack (**stackArray** and **topOfStack**). Changing the representation of **CharStack** would be difficult, because we would have to carefully inspect and modify all client modules. Other problems arise as well. The client module is responsible for initializing **topOfStack** to 0 before using the stack; this step is easy to overlook. Once it is initialized, we cannot guarantee the correct functioning of a **CharStack** variable, because the client module may modify **stackArray** or **topOfStack** in such a way as to corrupt the stack. (For example, suppose that the client accidentally increments **topOfStack** without storing a value in **stackArray**. The next call of **Pop** will fetch a meaningless character.)

We can improve **CharacterStacks** by making **CharStack** an opaque type, allowing **CharacterStacks** to export **CharStack** without providing any information to client modules about the actual structure of a **CharStack** variable. This approach solves our problems: we can change the representation of **CharStack** at any time without changing client modules, and we ensure that clients cannot modify **CharStack** variables except through procedures exported by **CharacterStacks**.

An improved version of the definition part of **CharacterStacks** follows, this time with **CharStack** defined as an opaque type. Note that **CharacterStacks** must export a procedure capable of initializing a stack; a client module has no direct way to initialize a **CharStack** variable, because the module cannot see the structure of the variable.

```

DEFINITION MODULE CharacterStacks;
  (* exports type CharStack (stack of characters) and
    procedures for manipulating variables of type CharStack *)

  TYPE CharStack; (* opaque export *)

  PROCEDURE Initialize(VAR s: CharStack);
    (* creates a stack s and makes it empty *)

  PROCEDURE Push(VAR s: CharStack; ch: CHAR);
    (* pushes ch onto stack s; terminates program if stack is full *)

  PROCEDURE Pop(VAR s: CharStack; VAR ch: CHAR);
    (* stores top element of stack s into ch, then pops s;
      terminates program if stack is empty *)

  PROCEDURE IsEmpty(s: CharStack): BOOLEAN;
    (* returns TRUE if s is empty, FALSE otherwise *)

END CharacterStacks.
```

The definition part of **CharacterStacks** gives no information about **CharStack** other than its name. Modules that import **CharStack** can declare variables of type

CharStack, but the only way to manipulate these variables is through the procedures **Initialize**, **Push**, **Pop**, and **IsEmpty**.

To show how a client module uses **CharacterStacks**, we will modify **Reverse** to use **CharacterStacks** instead of **CharStackModule**. **Reverse** must declare a variable **s** of type **CharStack**, initialize it by calling **Initialize**, and include **s** as a parameter when calling **Push**, **Pop**, and **IsEmpty**.

```

MODULE Reverse;
  (* reverses a string typed by the user *)

  FROM IO IMPORT RdKey, WrChar, WrStr, WrLn;
  FROM CharacterStacks IMPORT CharStack, Initialize, Push, Pop, IsEmpty;

  CONST cr = 15C;  (* carriage return *)

  VAR s: CharStack;
      ch: CHAR;

BEGIN
  Initialize(s);  (* make stack s empty *)

  WrStr("Enter string: ");
  ch := RdKey();
  WHILE ch # cr DO
    WrChar(ch);
    Push(s, ch);
    ch := RdKey();
  END;
  WrLn;

  WrStr("The reversal is: ");
  WHILE NOT IsEmpty(s) DO
    Pop(s, ch);
    WrChar(ch);
  END;
  WrLn;
END Reverse.

```

The call of **Initialize** is critical. Attempting to call **Push**, **Pop**, or **IsEmpty** before initializing **s** will have unpredictable results.

The full specification of the **CharStack** type must appear in the implementation part of **CharacterStacks**. Because **CharStack** must be a pointer type, we declare it to be a pointer to a record containing fields **stackArray** and **topOfStack**. The **Initialize** procedure allocates space for one of these records, then sets the **topOfStack** field to 0. Here is the implementation part of **CharacterStacks**:

```

IMPLEMENTATION MODULE CharacterStacks;

  FROM Storage IMPORT ALLOCATE;

  FROM IO IMPORT WrStr, WrLn;

  CONST stackSize = 100;  (* maximum size of a CharStack *)

```



```

TYPE StackRecord =
  RECORD
    stackArray: ARRAY [1..stackSize] OF CHAR;
    topOfStack: [0..stackSize+1]
  END;
CharStack = POINTER TO StackRecord;

PROCEDURE Initialize(VAR s: CharStack);
  (* creates a stack s and makes it empty *)
BEGIN
  ALLOCATE(s, SIZE(StackRecord));
  s^.topOfStack := 0
END Initialize;

PROCEDURE Push(VAR s: CharStack; ch: CHAR);
  (* pushes ch onto stack s; terminates program if stack is full *)
BEGIN
  INC(s^.topOfStack);
  IF s^.topOfStack > stackSize THEN
    WrStr("Stack overflow in CharacterStacks.Push");
    WrLn;
    HALT
  END;
  s^.stackArray[s^.topOfStack] := ch
END Push;

PROCEDURE Pop(VAR s: CharStack; VAR ch: CHAR);
  (* stores top element of stack s into ch, then pops s;
  terminates program if stack is empty *)
BEGIN
  IF s^.topOfStack = 0 THEN
    WrStr("Stack underflow in CharacterStacks.Pop");
    WrLn;
    HALT
  END;
  ch := s^.stackArray[s^.topOfStack];
  DEC(s^.topOfStack)
END Pop;

PROCEDURE IsEmpty(s: CharStack): BOOLEAN;
  (* returns TRUE if s is empty, FALSE otherwise *)
BEGIN
  RETURN s^.topOfStack = 0
END IsEmpty;

END CharacterStacks.

```

Separate Compilation

Modula-2 supports *separate compilation*; we can compile program modules and library modules separately. Separate compilation in Modula-2 differs from the *independent compilation* that some programming languages allow, where each program unit is compiled without knowledge of the other units. In Modula-2, the compiler checks that each entity imported from a library module is used in a manner consistent with its definition in the module. (The compiler performs this check using the information in the definition part of the library module; the implementation part of the library module plays no role.) When the implementation part of a library module is compiled, the compiler checks that it is consistent with the definition part of the module.

Separate compilation is a boon to the programmer: when we change a program, we need only recompile the affected modules, not every module in the program. Both separate and independent compilation have this property, but with separate compilation the compiler performs the same kind of type-checking among modules that it would do within a single module. These checks may reveal errors that would otherwise be difficult to find.

Compilation

The TopSpeed Modula-2 system assumes that each program module and implementation module is in a separate file with a `.MOD` extension and that each definition module is in a file with a `.DEF` extension. Most Modula-2 systems require that definition modules be compiled; however, in TopSpeed Modula-2, only program modules and implementation modules need be compiled (the order of compilation is not important). Compiling a program module or implementation module produces an `.OBJ` (object) file.

When we change a program module, we only need to recompile that one module (and re-link the entire program). Likewise, if we change the implementation part of a library module, and the changes do not affect the definition part, then we only need to recompile the implementation part. If we change the definition part of a module, however, we must recompile the implementation part and all modules that import from the library module. (If any of the affected modules is a definition module, we must recompile its clients as well.)

The process of recompiling a program after a change is simplified by the TopSpeed "Make" command. When the Make command is given, the system determines which modules need to be recompiled as the result of changes made to the program; it then recompiles these modules and relinks the program. See the *User's Manual* for details.

Linking

To link a Modula-2 program, a programmer need only invoke the TopSpeed linker (or use the Make command) and specify the name of the program module. The linker determines which modules to link to this module to complete the program. In general, the modules to be linked include not only modules from which the program module imports, but also modules imported by these modules, and so forth. The output of the linker is an `.EXE` (executable) file.

Version Checking

Modula-2 systems must check that the interfaces between modules are consistent even though the modules are compiled separately. TopSpeed Modula-2 uses a simple timestamp mechanism for this purpose.

Suppose that **M2.MOD** imports from library module **M1**. When we compile **M2.MOD**, the compiler will copy into **M2.OBJ** the date and time that **M1.DEF** was last modified. When we later compile **M1.MOD**, the compiler copies **M1.DEF**'s date and time to **M1.OBJ**. At linking time, the linker compares the date/time values stored in **M1.OBJ** and **M2.OBJ**; if they don't match, linking cannot proceed.

The purpose of this mechanism is to ensure that the code in **M1.OBJ** and **M2.OBJ** is consistent. For example, each call in **M2** of a procedure imported from **M1** should be consistent with the procedure declaration in **M1**. An inconsistency would arise if we changed **M1.DEF** and recompiled **M1.MOD** without recompiling **M2.MOD**. The compiler would copy **M1.DEF**'s new date and time into **M1.OBJ**, but **M2.OBJ** would still contain the old date and time, so **M1.OBJ** and **M2.OBJ** could not be linked. An error also occurs if we change **M1.DEF** and recompile **M2.MOD** without recompiling **M1.MOD**.

Local Modules

A program module, implementation module, or procedure, in addition to declaring constants, types, variables, and procedures, can declare modules as well. A module declared inside another module or procedure is called a *local module*. Declarations of local modules appear in the same place as declarations of the other entities; programmers can intermix the different kinds of declarations freely.

Local modules have properties similar to library modules, including the ability to import and export entities. However, local modules—unlike library modules—cannot be separately compiled.

Programming situations in which local modules are useful arise more often than one might expect. For example, consider the function **Random** (Chapter 5), which generates a sequence of pseudorandom **CARDINAL** numbers. We must save the value that **Random** computes, so that the next call of **Random** can use it. We cannot store this value in a variable local to **Random**, because then it would be lost when **Random** returns. Storing it in a variable **randomCardinal** that is global to **Random** leads to the version of **Random** given in Chapter 5:

```
PROCEDURE Random(): CARDINAL;
  (* returns a pseudorandom number between 0 and modulus - 1;
   * modifies the variable randomCardinal *)
BEGIN
  randomCardinal := (multiplier * randomCardinal + increment) MOD modulus;
  RETURN randomCardinal
END Random;
```

A potential problem appears here: the variable **randomCardinal**, because it is visible outside **Random**, is vulnerable to accidental change. By declaring a local module that contains both **randomCardinal** and **Random**, we can protect **randomCardinal**:


```

MODULE RandomNumberGenerator;
  (* exports modulus and Random; Random returns a pseudorandom
     CARDINAL number between 0 and modulus - 1 *)

  IMPORT seed;

  EXPORT modulus, Random;

  CONST multiplier = 40;
        increment = 3641;
        modulus = 729;

  VAR randomCardinal: CARDINAL;

  PROCEDURE Random(): CARDINAL;
    (* returns a pseudorandom number between 0 and modulus - 1;
       modifies the variable randomCardinal *)
  BEGIN
    randomCardinal := (multiplier * randomCardinal + increment)
                      MOD modulus;
    RETURN randomCardinal
  END Random;

BEGIN
  randomCardinal := seed
END RandomNumberGenerator;

```

The declaration of `RandomNumberGenerator` would appear in a program in place of the declaration of `Random`; we cannot compile `RandomNumberGenerator` separately. `RandomNumberGenerator` declares five entities: `multiplier`, `increment`, `modulus`, `randomCardinal`, and `Random`. Of these, only two—`modulus` and `Random`—are visible outside the module. (`RandomNumberGenerator` must export `modulus` to allow scaling of the value returned by `Random` into a desired range: 0.0 to 1.0 or 0 to 9, for example.) The other entities are not exported and thus are not visible outside `RandomNumberGenerator`. In particular, `randomCardinal` cannot be changed outside the module. (This is not strictly true in TopSpeed Modula-2, as we will see later.)

One problem with `RandomNumberGenerator` remains, however. `RandomNumberGenerator` imports the initial ("seed") value of `randomCardinal` and assigns it to `randomCardinal` in the body of the module. This method of initialization makes it difficult to restart the process of random number generation using a new seed value. To solve this problem, we add an initialization procedure to `RandomNumberGenerator`. We then export this procedure to make it visible outside the module:

```

MODULE RandomNumberGenerator;
  (* exports modulus and Random; Random returns a pseudorandom
     CARDINAL number between 0 and modulus - 1 *)

  EXPORT modulus, Random, InitializeSeed;

  CONST multiplier = 40;
        increment = 3641;
        modulus = 729;

  VAR randomCardinal: CARDINAL;

```



```

PROCEDURE Random(): CARDINAL;
  (* returns a pseudorandom number between 0 and modulus - 1;
   * modifies the variable randomCardinal *)
BEGIN
  randomCardinal := (multiplier * randomCardinal + increment)
                    MOD modulus;
  RETURN randomCardinal
END Random;

PROCEDURE InitializeSeed(seed: CARDINAL);
BEGIN
  randomCardinal := seed
END InitializeSeed;

END RandomNumberGenerator;

```

The module (or procedure) in which **RandomNumberGenerator** is declared can call **InitializeSeed** to restart random number generation with a different seed.

Local modules are particularly useful in conjunction with coroutines and interrupt handling, as we will see in Chapter 11. For example, local modules can be used to disable interrupts within selected regions of a program.

Local Module Declarations

A local module looks exactly like a program module, with two exceptions:

- A local module may contain an export list. If present, this must appear immediately after the import lists.

- A semicolon, not a period, follows the declaration of a local module.

By default, all entities declared in a local module are visible only within the module itself; entities visible in the enclosing block (module or procedure) are not visible in the module. In the following example, the variable **x** is visible throughout the **Outer** module but not in the **Inner** local module. The variable **y** is visible only inside **Inner**, not in **Outer**.

```

MODULE Outer;
  VAR x: REAL;

  MODULE Inner;
    VAR y: REAL;
    (* only y is visible *)
  END Inner;

  (* only x is visible *)
END Outer.

```

As a language extension, TopSpeed Modula-2 allows access to a local module's entities outside the module, provided that the name of each entity is qualified by the name of the module. For example, we can access the variable **y** outside **Inner** by writing **Inner.y**. This form of access is also allowed within **Inner** itself.

Import into a Local Module

An import list in a local module has the same form as an import list in a program module or implementation module. However, importing using the **FROM** option, which is common in program and implementation modules, is rare in local modules. Because identifiers visible in the block surrounding a local module are often not qualified, a local module can import them without the **FROM** option:

```
MODULE Outer;
  VAR x: REAL;

  MODULE Inner;
    IMPORT x;
    VAR y: REAL;
    (* both x and y are visible *)
  END Inner;

  (* only x is visible *)
END Outer.
```

The **FROM** option is useful when an entity is visible in the enclosing block only in qualified form. For example, if the module **Outer** imports **IO** using the import list

```
IMPORT IO;
```

all the entities exported by **IO** are visible in **Outer** in qualified fashion. If the local module **Inner** needs to import the procedure **WrStr**, it has two options: (1) import **IO** as a whole or (2) import **WrStr** alone using an import list with a **FROM** clause. The first option makes all the entities exported by **IO** visible in **Inner** (qualification is required, however). The second option makes only **WrStr** visible in **Inner**; no qualification is necessary.

Export from a Local Module

If the word **QUALIFIED** appears in a module's export list, we must use the name of the module to qualify the names of the exported entities:

```
MODULE Outer;
  VAR x: REAL;

  MODULE Inner;
    EXPORT QUALIFIED y;
    VAR y: REAL;
    (* only y is visible *)
  END Inner;

  (* both x and Inner.y are visible *)
END Outer.
```


Note: TopSpeed Modula-2 ignores export lists containing the word **QUALIFIED**, since all entities in the module are automatically accessible outside the module if qualified by the module name.

If we omit the word **QUALIFIED**, then we may reference exported entities outside the module by using their names alone. The following example shows the export of a variable without the **QUALIFIED** option.

```
MODULE Outer;
  VAR x: REAL;

  MODULE Inner;
    EXPORT y;
    VAR y: REAL;
    (* only y is visible *)
  END Inner;

  (* both x and y are visible *)
END Outer.
```

Module Initialization

A local module may have a *body*: a sequence of statements most often used to initialize the module's variables. In the following example, the body of the local module **Inner** initializes the variable **y**.

```
MODULE Outer;
  VAR x: REAL;

  MODULE Inner;
    EXPORT y;
    VAR y: REAL;
    BEGIN
      y := 1.0
    END Inner;

  BEGIN
    x := 2.0 * y + 1.0
  END Outer.
```

A program cannot "call" a local module in the way that it can call a procedure; instead, the body of a local module is executed automatically when the enclosing module or procedure is executed. The order of execution is important: The body of a local module is always executed immediately *before* the body of the enclosing module or procedure. In the example, the body of **Inner** will be executed before the body of **Outer**; thus, **y** has the value 1.0 when the body of **Outer** is executed. If several local modules are nested inside the same module or procedure, their bodies are executed in the order of the module declarations (but all are executed before the body of the enclosing module or procedure).

The body of a local module may contain a **RETURN** statement. The effect of executing a **RETURN** statement is to terminate execution of the module body. Execution continues

with the body of the module or procedure that would normally be executed next according to the rules just described.

The body of a local module declared in a program module or implementation module will be executed only once (because the body of the enclosing module is executed once). Variables in the local module are *static*: they retain their values throughout the execution of the program. However, the body of a local module declared in a procedure will be executed every time the procedure is called. In this case, the variables of the local module have the same lifetime as the variables of the procedure; the values of both are lost whenever the procedure returns. In general, variables in a local module have the same lifetime as variables in the enclosing block; thus, a local module restricts the visibility of variables without affecting their lifetime.

Chapter 10

Low-Level Programming

The first nine chapters of this tutorial describe the high-level, machine-independent features of Modula-2. Although these features are adequate for most applications, some programs require machine-level operations not normally supported by high-level programming languages. In particular, systems programs—assemblers, loaders, compilers, operating systems, and the like—must be able to

- fetch and store data using machine addresses
- access bits, bit fields, and bytes—units of storage that are not meaningful in most high-level languages
- perform machine-level operations (such as bitwise Boolean operations and shifting)
- access registers and buffers belonging to input/output devices

These "low-level" operations can be useful in programs for which fast execution or efficient use of space is critical.

Most high-level programming languages do not provide direct support for low-level programming. Instead, programmers must use "tricks" to accomplish the desired effect. (In Pascal, for example, programmers often use a record with a variant part to trick the compiler into thinking that a value of one type actually belongs to another type.) Wirth designed Modula-2 for systems programming, however, so it provides a rich assortment of low-level features:

The **BITSET** type allows access to individual bits in a word.

Types such as **WORD**, **BYTE**, and **ADDRESS**—and procedures such as **ADR**, which computes the address of a variable—support the use of words, bytes, and addresses.

Type transfer functions allow a program to treat a value of one type as though it belonged to another type.

The ability to locate a variable at a particular (absolute) address allows a program to access special memory locations.

Low-level programming has drawbacks. One problem, of course, is loss of portability. To achieve portability, programmers normally try to avoid features that depend on a particular computer or Modula-2 system. Low-level programming precludes complete portability. The other problem is the increased probability of errors. Operations on low-level values—words and addresses—are inherently error-prone; for example, storing a word into memory at the wrong address is an ever-present danger.

Careful use of Modula-2's low-level features can minimize the problems of low-level programming, however. Consider first the problem of transporting a program to a different environment (either a different computer, or a different Modula-2 system running on the same computer). The person transporting the program must locate any system or machine dependencies in the program and must modify them to fit the new environment. Modula-2 provides help for this task in two ways. First, the language's support for modules makes it easy for the original programmer to confine the use of low-level features to a few modules. Second, Wirth designed Modula-2 so that the reader of a program can easily determine where the original programmer used low-level features.

Programming at the machine level is normally done without types and therefore without the protection that type-checking provides. At the machine level, we have only bit patterns, which can be interpreted as numbers, characters, or other data values. The lack of types is one reason that programmers find it difficult to debug programs written in an assembly language. Modula-2 tries to solve this problem by enforcing normal type-checking throughout a program, requiring that the programmer explicitly inform the compiler at each point where it is necessary to suppress type-checking. Modula-2 compilers perform type-checking even for such low-level types as **WORD** and **ADDRESS**. Consider the type **WORD**, for example. Normally, it is impossible to perform arithmetic operations on **WORD** values. At the programmer's request, the compiler will treat a value of type **WORD** as though it were of another type, provided that values of that type occupy one word of memory. **CARDINAL** numbers usually require one word; by requesting that the compiler treat two **WORD** values as **CARDINAL** numbers, it becomes possible to add the values or perform other arithmetic operations on them. Because Modula-2 requires type-checking even for low-level types, a Modula-2 compiler can detect errors during compilation that would go undetected if the program had been written in assembly language or most other systems programming languages.

The BITSET Type

BITSET is the only standard (predefined) structured type in Modula-2; it has the following definition:

```
TYPE BITSET = SET OF [0..W-1];
```


The value of **W** may vary from one Modula-2 system to another, but is usually the number of bits in one word of memory. (In TopSpeed Modula-2, **W** is 16.)

BITSET has one property that distinguishes it from other set types: The word **BITSET** need not precede a value of type **BITSET** that appears in a program. Thus, the set value

```
{0, 8..15}
```

is the same as

```
BITSET{0, 8..15}
```

To make it easier to locate **BITSET** values in a program, the latter form is usually preferable.

BITSET is in the odd position of being a high-level type that is used primarily for low-level programming. By performing operations on a **BITSET** value, a program can examine and change the individual bits of a word. Most Modula-2 systems—including TopSpeed Modula-2—store a **BITSET** value in a single word (as the definition of **BITSET** suggests), where bit **i** of the word is 1 if **i** is a member of the set, and 0 if **i** is not a member.

This compact representation allows efficient storage and assignment of **BITSET** values. It also makes set operations easy to perform. For example, the set operation + (union) on two **BITSET** values corresponds to computing the bitwise OR of the two words that represent the sets. The other set operations also correspond to the bitwise Boolean instructions available on virtually all computers.

Knowing how a system represents **BITSET** values, the programmer can perform various low-level operations on words by treating them as **BITSET** values. The following table shows how we can interpret Modula-2's set operations—when applied to **BITSET** values—as operations on words:

w1 + w2	bitwise OR of words w1 and w2
w1 - w2	bitwise AND of w1 with complement of w2
w1 * w2	bitwise AND of words w1 and w2
w1 / w2	bitwise exclusive OR of words w1 and w2
{0..15} - w	bitwise complement of word w
i IN w	TRUE if bit i is set in word w
INCL(w, i)	set bit i in word w
EXCL(w, i)	clear bit i in word w

*Example Program: Sorting a File of **CARDINAL** Numbers*

It is sometimes possible to write efficient sorting programs that use a one-dimensional array of **BOOLEAN** values. For example, consider the problem of sorting a file of **CARDINAL** numbers in which no duplicates appear. If we declare the variable **bitVector** as

```
VAR bitVector: ARRAY [0..MAX(CARDINAL)] OF BOOLEAN;
```

then we can sort the file in the following way:

1. Set all components of **bitVector** to **FALSE**.
2. Read a number **i**, set **bitVector[i]** to **TRUE**, and repeat until the end of the file is reached.
3. Examine each component of **bitVector**, from first to last; if **bitVector[i]** is **TRUE**, write **i** to the output file.

If **CARDINAL** values are 16 bits long (thereby limiting **bitVector** to 65,536 components) and if the file of numbers we want to sort is large, this technique is often faster than conventional sorting methods. (Not counting the time taken to initialize **bitVector** and compare each component with **TRUE**, we can sort a file of N numbers in time proportional to N ; the best conventional methods require time proportional to $N \log N$.) The reason for its speed lies in the special characteristics of the file to be sorted: The values in the file come from a relatively small set (the **CARDINAL** numbers), there are no duplicate values, and no information is associated with each value (in most sorting applications, the sorting key is part of a record).

One problem with this algorithm is that the **bitVector** array occupies a great deal of space. In TopSpeed Modula-2, **MAX(CARDINAL)** is 65,535 and **BOOLEAN** values each occupy one byte, so the array requires 65,536 (64K) bytes.

To save space, we can pack 16 Boolean values into each word; **bitVector** then requires only 4,096 (4K) words (8K bytes). Because we need to access the individual bits in each word, we should declare **bitVector** to be an array of **BITSET** values:

```
CONST bitsPerWord = 16;
      vectorLength = MAX(CARDINAL) DIV bitsPerWord + 1;
VAR bitVector: ARRAY [0..vectorLength-1] OF BITSET;
```

Our sorting program needs to perform three operations on **bitVector**: Set bit **i** (where **i** is a number between 0 and 65,535), test whether bit **i** is set, and clear the entire vector. The procedures **SetBit**, **TestBit**, and **ClearVector**, respectively, perform these operations. **SetBit** and **TestBit** both use the fact that bit **i** of **bitVector** occurs in position **i MOD 16** of the component whose index is **i DIV 16**. **ClearVector** assigns the empty set to each component of **bitVector**. The full program follows.


```

MODULE SortCardinals;
  (* sorts a file of CARDINAL numbers; duplicate numbers are ignored *)

  FROM FIO IMPORT File, OK, Open, Create, Close, RdCard, WrCard, WrLn;

  CONST bitsPerWord = 16;      (* IBM PC *)
        vectorLength = MAX(CARDINAL) DIV bitsPerWord + 1;

  VAR bitVector: ARRAY [0..vectorLength-1] OF BITSET;
      i: CARDINAL;
      inFile, outFile: File;

  PROCEDURE SetBit(i: CARDINAL);
    (* sets bit i in bitVector *)
  BEGIN
    INCL(bitVector[i DIV bitsPerWord], i MOD bitsPerWord)
  END SetBit;

  PROCEDURE TestBit(i: CARDINAL): BOOLEAN;
    (* returns TRUE if bit i in bitVector is set *)
  BEGIN
    RETURN (i MOD bitsPerWord) IN bitVector[i DIV bitsPerWord]
  END TestBit;

  PROCEDURE ClearVector;
    (* clears every bit in bitVector *)
    VAR i: CARDINAL;
  BEGIN
    FOR i := 0 TO vectorLength - 1 DO
      bitVector[i] := BITSET{}
    END
  END ClearVector;

BEGIN
  (* read numbers and set corresponding bits in bitVector *)
  inFile := Open("IN.NUM");
  ClearVector;
  i := RdCard(inFile);
  WHILE OK DO
    SetBit(i);
    i := RdCard(inFile)
  END;
  Close(inFile);

  (* write the index of each 1 bit in bitVector *)
  outFile := Create("OUT.NUM");
  FOR i := 0 TO MAX(CARDINAL) DO
    IF TestBit(i) THEN
      WrCard(outFile, i, 1);
      WrLn(outFile)
    END
  END;
  Close(outFile)
END SortCardinals.

```

This program illustrates the use of the TopSpeed **FIO** (file I/O) module. To perform file I/O, we import **File** from **FIO** and use it to declare one or more file variables (**inFile** and **outFile** in this program). We can open a file by calling either **Open**, **Create**, or **Append** and supplying a file name. (**Open** opens an existing file for reading or writing. **Create** is similar, but creates the file if it does not already exist. **Append** allows data to be added to the end of an existing file.) All three functions return a value of type **File** (a *file handle*), which we store in a file variable for later use. The file handle

serves to identify the file in subsequent operations; for example, to read a **CARDINAL** number from a file, we must supply a file handle to the **RdCard** function. Once a file has been opened, we can read from it or write to it using procedures similar to those provided by the **IO** module. Files should always be closed (using the **Close** procedure) before program termination.

The **SYSTEM** Module

Every Modula-2 system must supply a module named **SYSTEM** that exports types and procedures that are useful for low-level programming. The exact contents of **SYSTEM** vary from one system to another, although **SYSTEM** normally exports the type **WORD**, whose values are single words of memory; the type **ADDRESS**, whose values are machine addresses; and the function **ADR**, which returns the address of a variable.

In TopSpeed Modula-2, values of type **WORD** are two bytes long and values of type **ADDRESS** are four bytes long. TopSpeed Modula-2 provides three other low-level types in addition to **WORD** and **ADDRESS**: **BYTE**, **LONGWORD** (four bytes), and **SHORTADDR** (16-bit address).

Note: In TopSpeed Modula-2, the types **WORD**, **ADDRESS**, **BYTE**, **LONGWORD**, and **SHORTADDR**, as well as the function **ADR**, need not be imported from **SYSTEM**; they are standard identifiers. (This is a language extension.)

The **SYSTEM** module typically provides much more than just low-level types. In TopSpeed Modula-2, **SYSTEM** exports a number of entities, including

- procedures for creating a coroutine and transferring control to a coroutine (see Chapter 11)
- procedures that enable or disable interrupts (see Chapter 11)
- procedures that allow direct communication with hardware ports

See the *User's Manual* for a complete description of **SYSTEM**.

SYSTEM does not reside in files, like a normal library module; instead, the compiler recognizes the entities that **SYSTEM** exports. Because of this property, **SYSTEM** is said to be a *pseudomodule*.

*The **WORD**, **BYTE**, and **LONGWORD** Types*

Values of type **WORD** represent single words of computer memory. In TopSpeed Modula-2, a value of type **WORD** contains 16 bits of information.

WORD values may be used as parameters, and functions may return a result of type **WORD**. The only operation defined for type **WORD** is assignment. (As a language extension, TopSpeed Modula-2 also permits testing two **WORD** values for equality or inequality.)

Modula-2 allows no other operations, including arithmetic and relational operations. However, type transfer functions (explained later in this chapter) make it possible to perform these operations on **WORD** values.

The primary usefulness of **WORD** lies in its special properties when used to declare formal parameters. If a formal parameter has type **WORD**, then the actual parameter can be of any type that occupies one word. (The list of allowable types varies from one system to another; in the TopSpeed Modula-2 system, one-word types include **CARDINAL**, **INTEGER**, **BITSET**, and **SHORTADDR**.) For example, we can use the following procedure to copy any one-word value to any one-word variable:

```
PROCEDURE CopyWord(source: WORD; VAR dest: WORD);
  (* copies any one-word value *)
BEGIN
  dest := source
END CopyWord;
```

Thus, if **b** and **i** are variables of types **BITSET** and **INTEGER**, respectively, the call

```
CopyWord(b, i);
```

will copy the contents of **b** to **i**, despite the fact that the variables have different types.

A formal parameter of type **ARRAY OF WORD** matches an actual parameter of *any* type, even an array or record type. An **ARRAY OF WORD** formal parameter allows us to treat the actual parameter as a block (array) of words; we can access individual words by indexing the formal parameter in normal array fashion. This feature of Modula-2 allows us to write procedures of great generality.

TopSpeed Modula-2 provides the types **BYTE** and **LONGWORD** in addition to **WORD**. A value of type **BYTE** contains eight bits of information; a value of type **LONGWORD** is four bytes (two words) long. The properties of the **BYTE** and **LONGWORD** types resemble those of **WORD**. If a formal parameter has type **BYTE**, the corresponding actual parameter can be any value that occupies one byte of storage. (Allowable types include **SHORT-INT**, **SHORTCARD**, **CHAR**, and **BOOLEAN**.) Similarly, declaring a formal parameter to be of type **LONGWORD** allows the actual parameter of any four-byte type (including **LONGCARD**, **LONGINT**, and **REAL**.) If a formal parameter has type **ARRAY OF BYTE** or **ARRAY OF LONGWORD**, a matching actual parameter can be of any type.

Note: In TopSpeed Modula-2, the types **WORD**, **BYTE**, and **LONGWORD** are assignment compatible with whole number types of equal size. For example, if **w** is of type **WORD** and **i** is of type **INTEGER**, both of the following statements are legal:

```
w := i;
i := w;
```

This is a language extension.

The ADDRESS and SHORTADDR Types and the ADR Function

A value of type **ADDRESS** is the address of a word in main memory. **ADDRESS** behaves as though it had the following declaration:

```
TYPE ADDRESS = POINTER TO WORD;
```

All the operations that Modula-2 allows for normal pointer values—assignment, comparison, and dereferencing—are also defined for values of type **ADDRESS**. For example, if we declare **addr** as

```
VAR addr: ADDRESS;
```

then **addr[^]** is the word in memory to which **addr** currently points.

ADDRESS also has special properties that other pointer types do not share. In particular, **ADDRESS** is compatible with any pointer type. Thus, if we declare **ptr** as

```
VAR ptr: POINTER TO T;    (* T can be any type *)
```

Modula-2 allows both of the following assignments:

```
ptr := addr;
addr := ptr;
```

Nevertheless, **ptr** and **addr** still point to values of different types: **ptr[^]** has type **T**, while **addr[^]** has type **WORD**. The compatibility of **ADDRESS** with all pointer types is particularly useful when we are writing procedures, because a formal parameter of type **ADDRESS** will match an actual parameter of any pointer type.

According to the Third Edition of Wirth's *Programming in Modula-2*, the **ADDRESS** type is compatible with some arithmetic type, such as **CARDINAL**, **INTEGER**, or **LONGINT**, and "all operators for integer arithmetic apply" to **ADDRESS** values. However, because of the IBM PC's segmented memory, TopSpeed Modula-2 requires that address arithmetic be done by calling procedures in the **Lib** module. The functions **AddAddr** and **SubAddr** allow a **CARDINAL** value to be added to an address or subtracted from an address. The procedures **IncAddr** and **DecAddr** allow us to increment or decrement a variable of type **ADDRESS**. See the *User's Manual* for more details.

As a language extension, TopSpeed Modula-2 provides the type **SHORTADDR**, which is compatible with any based pointer type. See the *User's Manual* for more information about based pointers.

If **x** is a variable, **ADR(x)** returns a value of type **ADDRESS** representing the address of **x** during program execution.

Example: Copying a Block of Words

As an example of the kind of low-level programming made possible by the **WORD** and **ADDRESS** types, consider the problem of writing a procedure to copy a block of words of arbitrary length from one address to any another address. We could not write this procedure using only the high-level features of Modula-2. If we use **WORD** and **ADDRESS**, however, the procedure is straightforward:

```
PROCEDURE CopyWords(source, dest: ADDRESS; length: CARDINAL);
  (* copies a block of words of the specified length from
     source address to dest address *)
  VAR i: CARDINAL;
BEGIN
  FOR i := 1 TO length DO
    dest^ := source^;          (* dest^ and source^ have type WORD *)
    IncAddr(source, SIZE(WORD)); (* IncAddr imported from Lib *)
    IncAddr(dest, SIZE(WORD))
  END
END CopyWords;
```

Suppose that a program needs to copy a block of words starting at address **addr** into the variable **buffer** (an array or record) for subsequent processing. The following call of **CopyWords** will perform the copy:

```
CopyWords(addr, ADR(buffer), SIZE(buffer) DIV SIZE(WORD));
```

The call of **ADR** returns the starting address of the variable **buffer**. The expression **SIZE(buffer) DIV SIZE(WORD)** computes the number of words that **buffer** occupies.

Note: The TopSpeed Modula-2 **Lib** module provides a procedure named **WordMove** that is similar to **CopyWords**, but more efficient.

Type Transfer Functions

By using types **WORD** and **ADDRESS**, we can treat any data value as an array of words and any pointer as an address. Modula-2 has a more general feature that allows us to treat a value of one type as though it were of another type, provided that the latter type has a name (call it **T**) and provided that values of the two types occupy the same amount of storage. We do so by using **T** as a function; a type name used in this fashion is called a *type transfer function*. For example, to treat a value of type **INTEGER** as though it had type **BITSET**, we can use the name **BITSET** as a function:

```
VAR i: INTEGER;
    s: BITSET;
    .
    .
    .
s := BITSET(i); (* view i as a BITSET *)
```


A call of a type transfer function has the same syntax as any function call; because it is an expression, not a designator, it can appear only where Modula-2 allows an expression. In particular, a call of a type transfer function cannot appear on the left side of an assignment statement.

A type transfer function does not actually transform its parameter. In our example, suppose that *i* has the value 89. Stored as a 16-bit word, *i* would look like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1

In TopSpeed Modula-2, the bits in a word are numbered from 15 (most significant) to 0 (least significant). Therefore, the value of *i*, interpreted as a **BITSET**, is {0, 3, 4, 6}. We did not alter the representation of *i* in any way, but simply interpreted it as a value of a different type. Clearly, proper use of type transfer functions requires knowledge of how a values of different types are represented in memory. (This information is available in the *User's Manual*.)

Modula-2 does not restrict type transfer to standard types; we may also use the names of programmer-defined types as type transfer functions. Furthermore, a type name used for type transfer need not represent an unstructured type; type transfer is legal for all types, including array and record types. Consider the following arrays; because they have different types, we cannot normally assign one to the other:

```
TYPE RealArray100 = ARRAY [1..100] OF REAL;
   RealArray101 = ARRAY [2..101] OF REAL;
VAR array100: RealArray100;
    array101: RealArray101;
```

Type transfer allows us to make the assignment, however, because the arrays require the same amount of storage:

```
array100 := RealArray100(array101);
```

The only requirement for applying a type name to an expression is that the value of the expression occupies the same amount of storage as do values of the type. This requirement, of course, is system-dependent; one compiler may reject a type transfer that another compiler allows.

Note: In TopSpeed Modula-2, type transfer is allowed even when values of the target type have a different size than the expression to which the transfer function is applied. See the *User's Manual* for details.

We must be careful to distinguish between type conversion and type transfer. Type *conversion* has the following characteristics:

- The data value we are converting may undergo an actual transformation; the bit pattern that results from the conversion usually differs from the original bit pattern.

- Type conversion often produces a value that occupies a different amount of storage (for example, a one-word **INTEGER** value may become a two-word **REAL** value).
- Type conversions can be checked for validity during program execution; an error occurs if the conversion would not produce a meaningful result.
- Type conversion is machine-independent. When we recompile a program on a different system, the compiler automatically supplies the correct algorithm for conversion; the programmer need not be aware of how the system represents data values.

Type conversion is performed by standard functions such as **FLOAT** and **TRUNC**, which convert between types **CARDINAL** and **REAL**. Type transfer has entirely different properties:

- Type transfer never causes a change in data representation.
- The result of a type transfer occupies the same amount of storage as the original value (it must, because the resulting bit pattern is identical to the original bit pattern).
- No check is performed to see if the result of a type transfer is meaningful; the programmer is responsible for using the result properly.
- Type transfer is system-dependent. When using type transfer, the programmer must know how a particular Modula-2 system represents data values. Programs that use type transfer usually require modification before they can be compiled and executed in a different environment.

Note: In TopSpeed Modula-2, a type transfer of the form **T (x)** is treated as a true type conversion if **T** is a basic type or an enumeration or subrange type and **x** is an expression whose type satisfies one of these conditions. For example, if **i** is an **INTEGER** variable and **r** is a **REAL** variable, the statement

```
i := INTEGER(r);
```

will convert **r** to type **INTEGER** and store the result in **i**.

Example: Bitwise Operations on Words

Recall that the only operation defined for values of type **WORD** is assignment. In order to perform any other operation, we must use type transfer to make the **WORD** value seem to be a **CARDINAL**, **INTEGER**, **BITSET**, or other value that occupies one word of memory. For example, suppose we want to compute the bitwise logical **OR** of two words. By using type transfer to make the words appear to be of type **BITSET**, we can use the set operator **+**:

```
PROCEDURE BitwiseOR(w1, w2: WORD): WORD;
  (* returns the bitwise OR of the words w1 and w2 *)
BEGIN
  RETURN WORD(BITSET(w1) + BITSET(w2))
END BitwiseOR;
```


BitwiseOR uses type transfer to make **w1** and **w2** seem to be of type **BITSET**; after computing the union of the sets, the procedure uses type transfer once more to make the result appear to be of type **WORD**. This procedure is very fast, because no genuine type conversion takes place.

Absolute Addresses

In some computers, certain memory locations have special meanings; for example, they may represent registers used to control input/output devices. Consequently, many Modula-2 compilers allow the programmer to specify that a variable must be located at a particular memory address, thus making it easier to access and/or modify the value stored at this address.

The mechanism for specifying the address of a variable is not a standard feature of Modula-2. On the IBM Personal Computer, an address consists of two 16-bit quantities: a *segment* and an *offset*. Consequently, TopSpeed Modula-2 requires that an absolute address be specified by two **CARDINAL** numbers—the segment and offset, in that order—separated by a colon.

Consider the problem of accessing certain locations in the memory of an IBM Personal Computer that contain information about the system's configuration. The *equipment flag*, stored at address **0:410H**, contains the following bit fields (among others):

Bit Field	Contents
15 – 14	number of parallel ports
11 – 9	number of serial ports
7 – 6	if bit 0 is set, number of diskette drives minus 1
5 – 4	initial video mode (01 = 40 x 25 color, 10 = 80 x 25 color, 11 = 80 x 25 monochrome)
0	set if system has diskette drives

(The most significant bit is numbered 15 and the least significant is numbered 0.) The word located at address **0:413H** contains the total size of main memory (in units of 1K).

To gain access to these memory locations, we can declare two variables, **equipFlag** and **memorySize**, and instruct the compiler to locate them at addresses **0:410H** and **0:413H**, respectively. To locate a variable at a particular address, we enclose the address in brackets and include it in the declaration of the variable:

```
VAR equipFlag [0:410H]: BITSET;
    memorySize [0:413H]: CARDINAL;
```

We may locate a variable of any type at an absolute address. We declare **equipFlag** to be a **BITSET** variable to allow access to individual bits in the word stored at address **0:410H**; we declare **memorySize** to have type **CARDINAL**, however, so that we can display the contents of the word at address **0:413H** as an integer.

Programmers should use the absolute address mechanism only for gaining access to memory locations with special meanings. Using it for other purposes (such as forcing variables to share memory locations) can make programs difficult to debug.

Example Program: Showing the Hardware Configuration of an IBM PC

The following program prints a summary of the information stored at addresses 0:410H and 0:413H.

```

MODULE ShowConfiguration;
  (* displays information about the hardware configuration of an IBM PC *)

  FROM IO IMPORT WrStr, WrCard, WrLn;

  CONST parMask      = BITSET{14..15};    (* parallel ports      *)
        parShift     = 14;
        serMask      = BITSET{9..11};    (* serial ports       *)
        serShift     = 9;
        diskMask     = BITSET{6..7};    (* diskette drives - 1 *)
        diskShift    = 6;
        videoMask    = BITSET{4..5};    (* initial video mode *)
        videoShift   = 4;
        driveBit     = 0;                (* set if system has  *)
                                         (* diskette drives    *)

  VAR equipFlag [0:410H]: BITSET;
      memorySize [0:413H]: CARDINAL;
      drives: CARDINAL;

  BEGIN
    WrStr("Number of parallel ports: ");
    WrCard(CARDINAL(equipFlag * parMask) >> parShift, 1);
    WrLn;

    WrStr("Number of serial ports: ");
    WrCard(CARDINAL(equipFlag * serMask) >> serShift, 1);
    WrLn;

    drives := 0;
    IF driveBit IN equipFlag THEN
      drives := CARDINAL(equipFlag * diskMask) >> diskShift + 1
    END;
    WrStr("Number of diskette drives: ");
    WrCard(drives, 1);
    WrLn;

    WrStr("Initial video mode: ");
    CASE CARDINAL(equipFlag * videoMask) >> videoShift OF
      1: WrStr("40 x 25 color") |
      2: WrStr("80 x 25 color") |
      3: WrStr("80 x 25 monochrome")
    END;
    WrLn;

    WrStr("Amount of memory installed: ");
    WrCard(memorySize, 1);
    WrStr("K");
    WrLn
  END ShowConfiguration.

```


This program uses the >> (right shift) operator, which is provided by TopSpeed Modula-2 as a language extension. The value of $i \gg j$ is i shifted j positions to the right; i and j must be of type **CARDINAL**, **SHORTCARD**, or **LONGCARD**. If this operator were not available, we could divide by a power of 2 instead. (For example, shifting right by 14 is the same as dividing by 214.)

Chapter 11

Coroutines and Interrupt Handling

Wirth designed Modula-2 to be a systems programming language; he intended to use Modula-2 to write the operating system (and all the other software, for that matter) for his Lilith computer. Consequently, Modula-2 supports the creation of programs (including—but not limited to—operating systems) that manage processes and respond to interrupts.

Modula-2's support for processes and interrupt-handling is based on the concept of the *coroutine*. We can simulate concurrent processes by using coroutines that transfer control among themselves, either directly or through a scheduling mechanism. We can also install coroutines as interrupt handlers; when an interrupt occurs, the appropriate coroutine services the interrupt, then suspends itself until the next interrupt occurs.

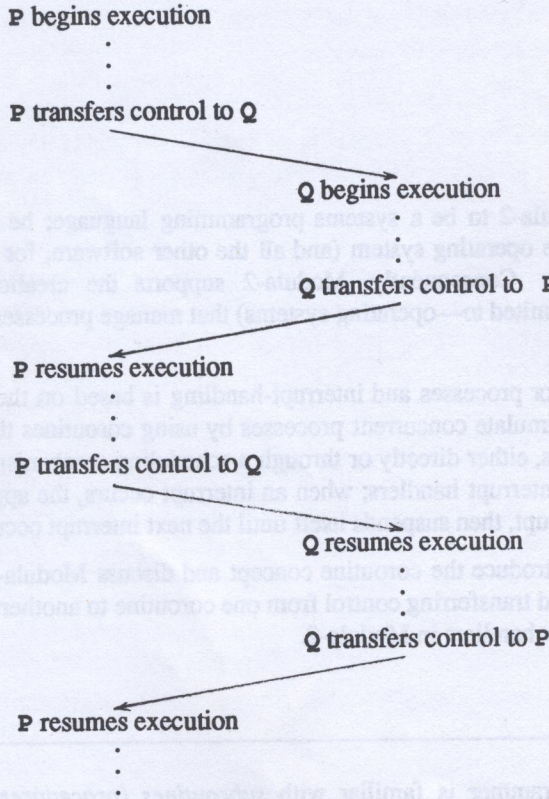
In this chapter, we introduce the coroutine concept and discuss Modula-2's features for creating coroutines and transferring control from one coroutine to another. We also show how to define interrupt handlers in Modula-2.

Coroutines

Although every programmer is familiar with *subroutines* (*procedures* in Modula-2), *coroutines* may be less familiar. Like a procedure, a coroutine contains both data (local variables) and code (statements). A procedure call activates a procedure; when the procedure returns control to its caller, the values of its local variables are lost. At the next call of the procedure, it begins to execute at the first statement in its body. We can activate a coroutine in much the same way we activate a procedure (we *transfer* control to the coroutine, however, rather than *call* it). When control leaves the coroutine, the values of its local variables are preserved. And, when we next activate the coroutine, it resumes execution at the statement immediately after the one that caused it to suspend execution

earlier. In other words, the system preserves the state of the coroutine's computation so that it resumes execution from where it left off.

Suppose that **P** and **Q** are coroutines, and a program transfers control to **P** for the first time. After executing for a while, **P** may transfer control to **Q**, which then begins executing. **Q** may subsequently transfer control back to **P**, which resumes executing where it left off earlier. (So far, **P** and **Q** have behaved no differently than if they were procedures.) If **P** later transfers control to **Q**, then **Q** resumes executing from the point at which it had transferred control to **P**; **Q** does not start at the beginning of its body. The flow of control between **P** and **Q** looks like this:



This example shows why we call **P** and **Q** *coroutines*: they are equal in status; neither is subordinate to the other. By comparison, if **P** and **Q** are procedures and **P** calls **Q**, then **P** cannot resume until **Q** has completed execution; here **Q** is subordinate to **P**.

A program may have two, three, or more coroutines. Each coroutine may transfer control to any of the other coroutines. It is not possible to have just one coroutine. When a program first transfers control to a coroutine, the program itself becomes—in effect—a

coroutine; the program and the coroutine coexist on an equal basis thereafter, with each capable of transferring control to the other.

Coroutines are useful in the following kinds of programs, among others:

- *Programs that consist of several components of equal status cooperating toward a common goal.* If no component is subordinate to the others, it may be easier to write the components as coroutines rather than procedures. For example, most compilers consist of several loosely connected components (lexical analyzer, parser, code generator, and so forth); it is natural to view these components as coroutines.
- *The kernel of an operating system.* One function of an operating system is to share the processor (or processors, in some computer systems) among many processes. The *kernel* performs this function by switching rapidly among different processes, allowing one to execute for a brief period before switching to another. The processes are essentially coroutines, because only one actually executes at a time. Furthermore, the state of each process is preserved while it is not executing. (Note, however, that processes—unlike coroutines in general—may be suspended involuntarily so that other processes have a chance to execute.)
- *Programs that must respond to events occurring outside the program.* Events may be internal to the system (a tick of the system clock, for example) or external (a keystroke generated at the system keyboard). Both cause an *interrupt*—a hardware signal that a particular event has occurred. Many Modula-2 systems allow a program to install a coroutine as an interrupt handler; the coroutine is activated automatically when the interrupt occurs, allowing the program to respond quickly to the event that caused the interrupt.
- *Programs that simulate discrete systems.* The programmer models each activity in a discrete system as a coroutine; by transferring control among the coroutines, the activities appear to progress in time.

Coroutine Creation

In Modula-2, we do not *declare* a coroutine; instead, we *create* it from a procedure. To create a coroutine, we call the **NEWPROCESS** procedure. (In the early development of Modula-2, Wirth used the term *process* instead of *coroutine*.) Coroutines in Modula-2 are inherently low-level—they require programs to manipulate **ADDRESS** values—so we must import **NEWPROCESS** from a low-level module (the **SYSTEM** module in TopSpeed Modula-2).

A call of **NEWPROCESS** must specify not only the procedure from which the coroutine is to be created, but also a *workspace* (a block of memory) in which to store the state of the coroutine:

```
PROCEDURE NEWPROCESS (P: PROC; A: ADDRESS; S: CARDINAL; VAR P1: ADDRESS);
```


The parameter **P** represents the procedure from which **NEWPROCESS** will create the coroutine. The parameters **A** and **S** indicate the address and size of the workspace. (In TopSpeed Modula-2, the size of a workspace is measured in bytes.) The fourth parameter, **P1**, must be a variable of type **ADDRESS**; **NEWPROCESS** stores a reference to the new coroutine in **P1**. The program will use the variable when the time comes for transferring control to the coroutine.

Modula-2 places two restrictions on the procedure **P** from which we create a coroutine. First, because **P** must be of type **PROC**, it cannot have parameters and cannot be a function. (Parameters are not as useful for coroutines as they are for procedures, because a coroutine does not begin executing from the beginning each time control is transferred to it. Instead, coroutines typically communicate through global variables.) Second, **P** must not be local to any other procedure. No other restrictions apply to **P**. In particular, **P** may contain local modules and may call procedures, including itself. (A coroutine that contains procedure calls may need a larger workspace than one that does not; we will return to this point at the end of the section.)

Before calling **NEWPROCESS** to create a coroutine, a program must allocate memory for the coroutine's workspace. A workspace is a region of memory that contains the local variables of the coroutine and the coroutine's stack, as well as other information. The simplest way to create a workspace is to declare an array of bytes (or words):

```
CONST wkspSize = 1000;
VAR wksp: ARRAY [1..wkspSize] OF BYTE;
```

Calculating an appropriate size for a workspace is difficult. We will sidestep the issue for the time being. A more detailed discussion of workspaces appears at the end of this section.

Before calling **NEWPROCESS**, a program must also declare an **ADDRESS** variable to reference the new coroutine:

```
VAR cor: ADDRESS; (* coroutine reference *)
```

A call of **NEWPROCESS** would then look like this:

```
NEWPROCESS (P, ADR(wksp), SIZE(wksp), cor);
```

Calling **NEWPROCESS** changes the value of **cor** to reference the new coroutine. Because **ADDRESS** values represent coroutines, programs do not depend on the actual representation of a coroutine, which may vary from one Modula-2 system to another.

We can create many identical coroutines from the same procedure. The procedure serves as a "template" for the coroutines we create from it; each of these coroutines executes the same statements and has local variables with the same names. However, each coroutine uses different memory locations for its local variables. The local variables belonging to a coroutine are stored in its workspace, and each coroutine has a different workspace.

Coroutine Transfer

The **TRANSFER** procedure allows one coroutine to transfer control to another:

```
PROCEDURE TRANSFER (VAR P1, P2: ADDRESS);
```

TRANSFER suspends the current coroutine (the one that called **TRANSFER**), stores a reference to it in **P1**, and resumes the coroutine that **P2** identifies (hence **P2** must have a value before we call **TRANSFER**, but **P1** need not). Like **NEWPROCESS**, **TRANSFER** must be imported from **SYSTEM**.

TRANSFER serves two purposes. First, it causes a newly created coroutine to begin executing:

```
NEWPROCESS (P, ADR(wksp), SIZE(wksp), new);  
TRANSFER (old, new);
```

The call of **TRANSFER** is necessary because **NEWPROCESS** never causes a coroutine to begin executing—it only prepares the coroutine to execute. Second, **TRANSFER** allows a previously suspended coroutine to resume executing. For example, to transfer control back to the coroutine that was executing before the call of **TRANSFER** just given, we would use the call

```
TRANSFER (new, old);
```

When a Modula-2 program calls **TRANSFER** for the first time, it essentially becomes a coroutine. Suppose that the first call of **TRANSFER** is

```
TRANSFER (main, cor);
```

where a call of **NEWPROCESS** has previously initialized **cor**. The main program is suspended, and the coroutine that **cor** identifies begins to execute. The coroutine executes until it calls **TRANSFER** or terminates. If it calls **TRANSFER**, it may choose to resume the main program (by supplying **main** as the second parameter to **TRANSFER**) or transfer control to some other coroutine. Once suspended, the main program is treated like any other coroutine—it remains suspended until another coroutine allows it to resume execution.

TRANSFER allows two coroutines to transfer control back and forth using a single **ADDRESS** variable. For example, suppose that the variable **previous** identifies a coroutine that is not currently executing. The call

```
TRANSFER (previous, previous);
```

suspends the currently executing coroutine and stores a reference to it in **previous** while allowing the previously executing coroutine to resume execution. This technique

works because **TRANSFER** first examines **previous** to determine which coroutine to resume, then changes **previous** to reference the newly suspended coroutine.

Coroutine Termination

If a coroutine ever reaches the end of its associated procedure (the point at which the procedure would normally return), the entire program—not just the coroutine—terminates, because the system cannot determine which coroutine should execute next. Therefore, procedures from which programs will create coroutines normally either contain an infinite loop (ensuring that the end of the procedure is never reached) or end with a call of **TRANSFER**:

```
PROCEDURE P;    (* to be used to create a coroutine *)
.
.
.
BEGIN
.
.
.
    TRANSFER(current, main); (* return control to main program *)
END P;
```

Of course, if the coroutine is ever resumed after this final transfer, it will reach the end of the procedure and the entire program will halt.

A coroutine is not an *entity*, because it is not declared and does not have a name. A coroutine is more like an anonymous variable—it is created dynamically and is "pointed to" by a variable. No mechanism exists for "disposing of" a coroutine (we can dispose of an anonymous variable by calling **DEALLOCATE**). However, deallocating the coroutine's workspace essentially destroys the coroutine as well. Attempting to transfer control to a coroutine whose workspace no longer exists will have unpredictable effects.

Example Program: Printing Fibonacci Numbers

As an example of how to use coroutines, we present a simple program named **PrintFibonacciNumbers** that prints all Fibonacci numbers up to some predetermined limit. (A similar program—without coroutines—appears in Chapter 1.)

PrintFibonacciNumbers uses two variables, **fib1** and **fib2**, to keep track of the last Fibonacci number printed and the one to print next. Two coroutines share the tasks of printing numbers and updating **fib1** and **fib2**. When **fib1** contains the next Fibonacci number, one coroutine prints **fib1**, adds **fib1** to **fib2** (so that **fib2** contains the next Fibonacci number), then transfers control to the other coroutine, which is identical except that it reverses the roles of **fib1** and **fib2**.

The two coroutines appear in the program as procedures named **IncFib1** and **IncFib2**. The body of **PrintFibonacciNumbers** creates coroutines from **IncFib1** and **IncFib2**, using the following calls of **NEWPROCESS**:

```
NEWPROCESS (IncFib1, ADR(wksp1), SIZE(wksp1), cor1);
NEWPROCESS (IncFib2, ADR(wksp2), SIZE(wksp2), cor2);
```

The variables **cor1** and **cor2** now contain references to the coroutines created from **IncFib1** and **IncFib2**, respectively. **PrintFibonacciNumbers** next transfers control to **IncFib1**:

```
TRANSFER(main, cor1);
```

TRANSFER saves a reference to the main program in the variable **main** before it transfers control to **IncFib1**. The following actions then take place:

1. **IncFib1** begins to execute. It writes 1 (the value of **fib2**), adds **fib2** to **fib1** (**fib1** is now 1), then transfers control to **IncFib2**, using the reference in **cor2**.
2. **IncFib2** begins to execute. After writing 1 (the value of **fib1**), it adds **fib1** to **fib2** (**fib2** is now 2), then transfers control to **IncFib1**, using the reference in **cor1**.
3. **IncFib1** resumes execution. It writes 2 (the value of **fib2**), adds **fib2** to **fib1** (**fib1** is now 3), then transfers control to **IncFib2**.
4. **IncFib2** resumes execution. After writing 3 (the value of **fib1**), it adds **fib1** to **fib2** (**fib2** is now 5), then transfers control to **IncFib1**.

Execution continues this way, with control alternating between **IncFib1** and **IncFib2**, until one of the coroutines detects that the value it was about to print exceeds **maxFib**. It then transfers control back to the main program, using the reference stored in the variable **main**.

PrintFibonacciNumbers looks like this:

```
MODULE PrintFibonacciNumbers;
  (* prints Fibonacci numbers using coroutines *)

  FROM SYSTEM IMPORT NEWPROCESS, TRANSFER;
  FROM IO IMPORT WcCard, WcLn;

  CONST wkspSize = 1000;
        maxFib = 10000;    (* prints numbers up to this limit *)

  VAR wksp1, wksp2: ARRAY [1..wkspSize] OF BYTE;
      main, cor1, cor2: ADDRESS;    (* coroutines *)
      fib1, fib2: CARDINAL;
```



```

PROCEDURE IncFib1;
(* prints fib2, then adds fib2 to fib1 *)
BEGIN
  WHILE fib2 <= maxFib DO
    WrCard(fib2, 1);
    WrLn;
    INC(fib1, fib2);
    TRANSFER(cor1, cor2)
  END;
  TRANSFER(cor1, main)
END IncFib1;

PROCEDURE IncFib2;
(* prints fib1, then adds fib1 to fib2 *)
BEGIN
  WHILE fib1 <= maxFib DO
    WrCard(fib1, 1);
    WrLn;
    INC(fib2, fib1);
    TRANSFER(cor2, cor1)
  END;
  TRANSFER(cor2, main)
END IncFib2;

BEGIN
  fib1 := 0;
  fib2 := 1;
  NEWPROCESS(IncFib1, ADR(wksp1), SIZE(wksp1), cor1);
  NEWPROCESS(IncFib2, ADR(wksp2), SIZE(wksp2), cor2);
  TRANSFER(main, cor1)
END PrintFibonacciNumbers.

```

Workspaces

So far, we have ignored two important questions about coroutine workspaces: how large should a workspace be, and where should we locate it?

To answer the first question, we need to know what a workspace contains. Although the exact contents vary from one Modula-2 system to another, a workspace usually includes the following:

- a fixed-size area in which to save the state of the coroutine when it is suspended
- the coroutine's local variables
- the coroutine's stack (used for procedure calls performed while the coroutine is executing)

In TopSpeed Modula-2, a workspace should usually be at least 1,000 bytes long. A coroutine may need a larger workspace if it calls procedures, because storage for procedure calls is allocated on the coroutine's stack.

We can put a workspace in one of three places:

- in the space allocated for global variables
- on the system stack, which contains space for local variables belonging to procedures

in the system heap, which contains space for anonymous, dynamically allocated variables

Which location we use depends on the lifetime of the coroutine. If a coroutine will have the same lifetime as the program itself, then we should store its workspace with the program's global variables:

```
MODULE M;
  CONST wkspSize = 1000;
  VAR workspace: ARRAY [1..wkspSize] OF BYTE;
  .
  .
END M;
```

If a coroutine exists only while a certain procedure is active, then it makes sense to store the workspace with that procedure's local variables. In this way, the workspace only occupies space while the procedure is executing.

```
PROCEDURE P;
  CONST wkspSize = 1000;
  VAR workspace: ARRAY [1..wkspSize] OF BYTE;
  .
  .
END P;
```

The third possibility is that we will be creating and destroying coroutines in a fairly arbitrary fashion. It is then best to store workspaces in the system heap, because space in the heap can be allocated and deallocated under program control.

```
VAR workspace: ADDRESS;
.
.
.
ALLOCATE(workspace, wkspSize);
```

When using stack or heap storage for a workspace, the programmer must ensure that the workspace remains in existence throughout the coroutine's lifetime. Deallocating the workspace while the coroutine still exists will have unpredictable (and usually disastrous) effects.

Another factor that affects workspace placement is whether the number of coroutines that a program will create is fixed or will vary from one execution of the program to another. If the number can vary, then the program must either declare workspaces as local variables in a recursive procedure (each recursive call of the procedure allocates room for one or more new workspaces) or—more often—allocate them dynamically in the heap.

Interrupt Handling

Most single-processor systems provide a limited form of true concurrency—they allow a program to execute at the same time that input/output operations are taking place. Of course, the program must pause occasionally to allow data transfer to or from an I/O device; many computer systems provide *interrupts* for this purpose. An interrupt is a hardware signal generated when a particular event occurs (such as the completion of an I/O operation). The procedure **IOTRANSFER** allows programs to detect and respond to interrupts. (Although Wirth states in *Programming in Modula-2* that **IOTRANSFER** is specific to the PDP-11 computer, Modula-2 systems for other computers often support it.)

Using **IOTRANSFER**, a program can suspend a coroutine until a specified interrupt occurs. The coroutine serves as an *interrupt handler*—when control is transferred to it as the result of an interrupt, the coroutine does whatever is necessary to respond to the interrupt. By calling **IOTRANSFER** again, the coroutine can then return control and suspend itself until the next interrupt.

Three steps are involved in creating an interrupt handler. First, the program must create a coroutine (by calling **NEWPROCESS**) to serve as the interrupt handler. Second, the program must call **TRANSFER** to transfer control to the interrupt handler. After the interrupt handler has initialized its local variables (and perhaps initialized the device that it manages), it performs the third step by calling **IOTRANSFER**, which suspends the interrupt handler until a particular interrupt occurs. The call of **IOTRANSFER** must specify, by giving an *interrupt vector number*, which interrupt the handler should wait for. The call of **IOTRANSFER** must also specify what coroutine to execute after the interrupt handler is suspended.

IOTRANSFER has the following heading:

```
PROCEDURE IOTRANSFER(VAR P1, P2: ADDRESS; I: CARDINAL);
```

The variables **P1** and **P2** are references to coroutines, while **I** is an interrupt vector number. In TopSpeed Modula-2, **IOTRANSFER** must be imported from the **SYSTEM** module.

A call of **IOTRANSFER** suspends the current coroutine (the interrupt handler), stores a reference to this coroutine in **P1**, and allows the coroutine referenced by **P2** to resume execution. In addition, **IOTRANSFER** "installs" **P1** as the handler for the interrupt specified by **I**. When this interrupt next occurs, the following actions take place:

1. The current coroutine is suspended.
2. A reference to this coroutine is stored in **P2**.
3. The coroutine referenced by **P1** (the interrupt handler) resumes execution.

These actions are identical to those performed for an ordinary **TRANSFER**; the interrupt essentially causes the insertion of a call of **TRANSFER** wherever the program happens to be when interrupted.

After responding to an interrupt, an interrupt handler executes **IOTRANSFER** to suspend itself until the next interrupt occurs and at the same time to return control to the interrupted coroutine. This call of **IOTRANSFER** is usually identical to the call that originally installed the interrupt handler, suggesting that we should enclose **IOTRANSFER** in a **LOOP** statement. In fact, most interrupt handlers have the following appearance:

```
PROCEDURE InterruptHandler;
  (* declarations of local variables *)
BEGIN
  (* initialize local variables *)
  LOOP
    IOTRANSFER(handler, mainProcess, interruptVector);
    (* respond to interrupt *)
  END
END InterruptHandler;
```

To initialize the interrupt handler, the main program creates a coroutine from **InterruptHandler** and transfers control to it:

```
NEWPROCESS(InterruptHandler, ADR(wksp), SIZE(wksp), handler);
TRANSFER(mainProcess, handler);
```

We can summarize the actions that take place from this point as follows:

1. Control is transferred to the interrupt handler.
2. The interrupt handler initializes its local variables, then calls the **IOTRANSFER** procedure.
3. **IOTRANSFER** installs the interrupt handler, then suspends it and transfers control to the main program.
4. When the specified interrupt occurs, the currently executing coroutine is suspended and assigned to the variable **mainProcess**; the handler resumes from the point just after the **IOTRANSFER** operation. When control reaches the top of the **LOOP** statement, the call of **IOTRANSFER** reinstalls the interrupt handler and transfers control to the interrupted coroutine (referenced by **mainProcess**).

IOTRANSFER can be hard to understand because it serves two distinct purposes: installing an interrupt handler and resuming an interrupted coroutine. These actions are so similar that Modula-2 combines them into one procedure.

Module Priorities

It is important to ensure that an interrupt handler is not itself interrupted by the same interrupt (or, in some cases, by other interrupts). Modula-2 allows the programmer to dis-

able interrupts within a module by specifying a *priority* in the heading of that module. A priority is a constant expression of type **CARDINAL** enclosed in brackets. A priority can appear in the heading of a program module, implementation module, or local module. (A definition module cannot be given a priority, because definition modules contain no statements and therefore cannot be interrupted.)

On some computers (including the PDP-11, the machine on which Modula-2 was first implemented), each interrupt has an associated priority. Modula-2 systems for such computers interpret the priority in a module heading as a command to disable interrupts of equal or lower priority. Thus, when control enters a module whose heading specifies a priority value of n , interrupts with priority n or less are disabled. When control leaves the module, the previous priority is restored. (Control enters a module when its body is executed or one of its procedures is called.)

Procedures within a module of priority n may call procedures outside the module. These procedures may be in modules of equal priority, greater priority, or no priority. (Modula-2 forbids calling a procedure in a module of lower priority.) When control enters a module for which the programmer has specified no priority, the priority level remains unchanged. (For example, if a procedure in a module with priority 5 calls a procedure in a module with no priority, interrupts of priority 5 or less remain disabled.) Calling a procedure in a module of higher priority (m , say) causes interrupts with priority m or less to be disabled temporarily; when the procedure returns, the priority returns to its old value.

In the IBM PC family, interrupts do not have an associated priority. Instead, the interrupt controller contains a mask that indicates which interrupts are currently disabled. In Top-Speed Modula-2, the priority in a module heading, although it is a **CARDINAL** value, is viewed as a **BITSET** mask whose elements indicate which interrupts should be disabled when control enters the module. For more details, consult the *User's Manual*.

Every module that contains a call of **IOTRANSFER** must be given a priority. However, modules that have a priority should be kept small to avoid losing interrupts unnecessarily. Typically, programmers attach priorities to local modules rather than to entire compilation units.

Index

- ABS** (standard function), 35, 37
- Absolute addresses**, 18, 164–166
- Abstraction**, 129
- ACos** (**MATHLIB** function), 38
- Active variant**, 104–106
- Actual parameter list**, 67
 - empty, 67, 74–75
 - of a function, 74–75
- Actual parameters**, 10, 67, 70–73
- Ada**, 1
- Addition**, 34, 36
- Addition operators**, 44
- ADDRESS** type, 17, 160
- Addresses**, 18, 160, 164–165
- ADR** (standard function), 17, 160
- Aggregates**, 46
 - array, 92–93
 - record, 100
- Alias declarations**, 47
- ALLOCATE** (**Storage** procedure), 115–116
- Ampersand**, 8, 43
- AND** (logical operator), 8, 43
- Anonymous types**, 79–80, 87
- Anonymous variables**, 7, 114–120
 - creating, 115–116
 - disposing of, 119–120
- Append** (**FIO** function), 157
- Arguments**, *see* Parameters
- Arithmetic expressions**, 33–39
 - See also* Expressions
- Arithmetic operators**, 8
- Arithmetic types**, *see* Numeric types
- Array aggregates**, 92–93
- ARRAY OF BYTE** parameters, 159
- ARRAY OF LONGWORD** parameters, 159
- ARRAY OF WORD** parameters, 159
- Arrays**, 6–7, 91–96
 - assignment of, 93, 94
 - components of, 91–92
 - component type of, 91–92
 - index type of, 6, 91–92
 - multidimensional, 93–94
 - open, 10, 71, 95–96
- Array types**, 6–7, 77–78, 91–96
 - specification of, 92
- ASCII** (American Standard Code for Information Interchange), 39, 40, 41
- ASin** (**MATHLIB** function), 38
- Assignment**
 - of arrays, 93, 94
 - of enumeration values, 82
 - of pointers, 117–118
 - of procedures, 124–125
 - of records, 100
 - of sets, 108
 - of strings, 97–98
 - of variables of an opaque type, 140
- Assignment compatibility**, 51, 88, 89
- Assignment statement**, 8, 50–52
- Associativity**, operator, 43, 44
- ATan** (**MATHLIB** function), 38
- ATan2** (**MATHLIB** function), 38

Available (Storage function),
116

Based pointers, 114, 160

Base type

of a set type, 107

of a subrange type, 85

Basic types, 33–47

See also **BOOLEAN; CARDI-**

NAL; CHAR; INTEGER;

LONGCARD; LONGINT;

LONGREAL; REAL;

SHORTCARD; SHORTINT

BITSET (standard type), 17, 107,
154–158

Blocks, 30

Body

of an implementation module,
132, 138

of a local module, 150–151

of a loop, 56

of a procedure, 66

BOOLEAN (standard type), 42–44,
81

BOOLEAN expressions, 8, 43–44

Braces, 108

Brackets, 6, 85, 92

B suffix, for octal numbers, 4, 23

BY clause, in a FOR statement, 9

BYTE (standard type), 159

C (programming language), 1

Call

of a function, 74–75

of a procedure variable,
125–126

of a proper procedure, 67

CAP (standard function), 41

CARDINAL (standard type), 3,
35–36

converting to and from **INTE-**
GER, 38

Caret, 117

Case expression, 9, 55

Case labels

in a **CASE** statement, 9, 55

in the variant part of a record,
105

Cases, in a CASE statement, 55–56

Case sensitivity, 3, 4, 22

CASE statements, 9, 54–56

CHAR (standard type), 39–42

Character literals, 39, 40

Characters, 39–42

control, 40

nonprinting, 40

Character strings, see Strings

CHR (standard function), 40

Circular import, in implementation
modules, 140

Client module, 131

Close (FIO procedure), 158

Comments, 5, 26

Compatibility, 37, 87–88, 89

Compilation

of definition modules, 28

of implementation modules,
145

independent, 144–145

order, 145

of program modules, 145

separate, 15, 130, 144–146

Compilation units, 15

Compiler directives, 26, 85, 92,
118

Compilers, one-pass, 69–70

Components

of an array, 91–92

of a record, 99–100

Component type, of an array,
91–92

Concatenation of strings, 25

Constant declarations, 46–47

Constant definitions, 133

Constant expressions, 8, 45

- Constants,
 - set, 109
 - standard, *see* Standard constants
 - See also* Literals
- Control characters, 40
- Control structures, 9, 49
 - See also* CASE statements; FOR statements; IF statements; LOOP statements; REPEAT statements; WHILE statements
- Control variable, in a FOR statement, 61–63
- Conversion, type, 37–38, 162–163
- Coroutines, 18, 167–175
 - creation, 169–170
 - as interrupt handlers, 176–177
 - termination, 172
 - uses of, 169
- Coroutine transfer, 167–168, 171–172
- Coroutine workspaces, 169–170, 174–175
- Cos (MATHLIB function), 38–39
- Cosh (MATHLIB function), 38
- Create (FIO function), 157
- C suffix, for character literals, 4, 40
- Dangling pointers, 119–120
- Data types, *see* Types
- DEALLOCATE (Storage procedure), 119
- DEC (standard procedure), 51–52, 83–84
- Declarations, 5, 30, 67–68
 - alias, 47
 - constant, 46–47
 - function, 73–74
 - in implementation modules, 136–138
 - label, 5, 30
 - module, 148
 - order of, 5, 30, 69–70
 - procedure, 65–70
 - in procedures, 67–68
 - type, 79–80
 - variable, 47, 79
- DEC PDP-11, 176, 178
- Definite iteration, 49, 62
- Definition modules, 12, 15, 130–133, 133–136
 - compilation of, 28
 - import lists in, 134
 - See also* Modules
- Definitions, 130–131, 133–134
 - constant, 133
 - procedure, 134
 - type, 133
 - variable, 134
- Dereferencing, 116–118
- Designators, 50–51, 74, 92, 101–102
- Difference, set, 109
- Direct recursion, 69–70
- DIV (operator), 34
- Division, 36
- Double quote character, 24, 39, 97
- Dynamic storage allocation, 114, 115–116
- E (scale factor), 24
- Elements, set, 108
- ELSE clause
 - in a CASE statement, 9, 55
 - in an IF statement, 53–54
 - in the variant part of a record, 106
- ELSIF clauses, in an IF statement, 54
- Empty case, 56
- Empty parameter list, 67, 74–75
- Empty set, 106, 108
- Empty statement, 31
- Empty variant, 106
- Entities, 30

Enumeration types, 6, 77–78,
78–79, 80–84

export of, 133

specification of, 81

writing values of, 84

Equality

of arrays, 93

of pointers, 119

of procedures, 124

of records, 101

of sets, 110–111

of strings, 97

of variables of an opaque type,
140

Equivalence, type, 86–87, 89

Example programs

computing the circumference
and area of a circle, 47

converting a Fahrenheit tem-
perature to Celsius, 39

encoding a message, 41

integer exponentiation, 3–4

printing Fibonacci numbers,
172–174

showing the hardware configu-
ration of an IBM PC,
165–166

sorting a file of **CARDINAL**
numbers, 156–158

testing the **MATHLIB** module,
60–61

Examples

bitwise operations on words,
163–164

checking weekly schedules,
112

copying a block of words, 161

the definition part of a stack
module, 134–136

the implementation part of a
stack module, 138–139

a module that exports an
opaque stack type,
141–144

updating inventory records,
101

using pointers to construct
linked lists, 120–123

writing values of an enumera-
tion type, 84

EXCL (standard procedure),
111–112

Executable files, 145

Exit from the middle of a loop,
58–59

Exit points, multiple, 59

EXIT statements, 9, 50, 58–59

Exp (**MATHLIB** function), 14, 38

Export, 11, 129

of enumeration types, 133

from a local module, 149–150

opaque, 140

of record types, 133

qualified, 149–150

transparent, 133

Export lists

in local modules, 149–150

QUALIFIED option, 149–150

Expressions, 8, 44–45

arithmetic, 33–39

constant, 8, 45

logical, 8, 43–44

mixed, 37–38

set, 109–111

FALSE (standard constant), 42–43

Field lists, 105

Field names, scope of, 99–100

Fields, 99–101

assignment to, 100

tag, 103–106

File (**FIO** type), 157

File handles, 157–158

Files, 157–158

- appending, 157
- closing, 158
- creating, 157
- opening, 157

File variables, 157**FIO (library module), 10–11,**
29–30, 157–158**FLOAT (standard function), 37–38****Formal parameter list, 66, 71**

- empty, 74–75
- of a function, 74–75

Formal parameters, 66, 70–73

- of type **ADDRESS**, 160
- of type **ARRAY OF BYTE**, 159
- of type **ARRAY OF LONG-WORD**, 159
- of type **ARRAY OF WORD**, 159
- of type **BYTE**, 159
- of type **LONGWORD**, 159
- of type **WORD**, 159

FOR statements, 4, 9, 61–63, 83

- control variable, 61–63
- increment, 62–63
- limit, 61–63
- starting value, 61–63

FORWARD, 70**Forward references**

- in procedure declarations, 69–70
- in the specification of pointer types, 5, 30, 79, 120–121

FROM clause, in an import list,
28–29, 149**Function declarations, 73–74****Function designators, 74****Function heading, 73****Function procedures, *see* Functions****Functions, 10, 65, 73–75**

- calling, 74–75
- parameter list of, 74–75
- recursive, 74

result type of, 74, 118–119,
126–127

standard, *see* Standard functions

See also Procedures

Garbage, 119**Global entities, 67–68****GOTO statements, 5, 8–9, 50****HALT (standard procedure), 9, 50****Heading**

- function, 73
- procedure, 66

Heap storage, 175**Hexadecimal numbers, 4, 23–24****Hierarchy, operator, *see* Operator precedence****HIGH (standard function), 95****H suffix, for hexadecimal numbers,**
4, 23**IBM Personal Computer**

- addresses, 160, 164
- equipment flag, 164
- interrupts, 178

Identifiers, 4, 21–22

- qualified, 29
- qualifying, 29
- standard, *see* Standard identifiers

IF statements, 9, 52–54**Implementation modules, 12, 15,**
130–133, 136–139

- body of, 132, 138
- circular import in, 140
- compilation of, 145
- declarations in, 136–138
- import lists in, 136–137
- See also* Modules

Import, 4, 11, 28–30

- circular, 140
- into a definition module, 134

- into an implementation module, 136–137
 - into a local module, 149
- Import lists, 28–30
 - in definition modules, 134
 - FROM** clause in, 28–29, 149
 - in implementation modules, 136–137
 - in local modules, 149
- Improper subset, 111
- IN** (relational operator), 42–43, 110
- INC** (standard procedure), 17, 51–52, 83–84
- INCL** (standard procedure), 111–112
- Increment, in a **FOR** statement, 62–63
- Indefinite iteration, 49, 62
- Independent compilation, 144–145
- Index expression, 92
- Index type, of an array, 6, 91–92
- Indices, array, 91
- Indirect recursion, 69–70
- Inequality
 - of arrays, 93
 - of pointers, 119
 - of procedures, 124
 - of records, 101
 - of sets, 110–111
 - of strings, 97
 - of variables of an opaque type, 140
- Infinite loop, 58
- Infix call
 - of a function, 75
 - of a proper procedure, 67
- Information hiding, 130
- Initialization order, of modules, 139–140, 150
- Input/output, 4, 10–11
- Input/output modules, *see* **FIO**; **IO**
- INTEGER** (standard type), 34–35
 - converting to and from **CARDINAL**, 38
- INTEGER** operators, 34–35
- Integers, 4, 23–24
- Interrupt handlers, 18–19, 169, 176–177
- Interrupt handling, 18–19, 176–178
- Interrupt priority, 178
- Interrupts, 18–19, 169, 176–178
 - disabling, 19, 177–178
- Interrupt vectors, 176
- Intersection, set, 109
- IO** (library module), 4, 10, 28–30, 41
- IOTRANSFER** (**SYSTEM** procedure), 19, 176–178
- Iteration, 49, 62
- Kernel of an operating system, 169
- Labels, 5, 30
- Left shift (operator), 8, 25
- Lib** (library module), 161
- Library modules, 11, 12–14, 27–28, 129, 130–144
 - definition part, *see* Definition modules
 - implementation part, *see* Implementation modules
 - See also* Modules
- Lifetime, of variables, 68, 138, 151
- Lilith, 1
- Limit, in a **FOR** statement, 61–63
- Linked lists, 120–123
 - defining, 120
 - deleting a node, 122–123
 - inserting a node, 121–122
- Linked structures, *see* Linked lists
- Linking, 15, 145
 - version checking during, 145–146
- Literals
 - character, 39, 40

- integer, 4, 23–24
- real, 24
- string, 24–25, 39, 97–98
- Log** (**MATHLIB** function), 14, 38
- Log10** (**MATHLIB** function), 38
- Local entities, 67–68
- Local modules, 12, 16–17, 129, 138, 146–151
 - body of, 150–151
 - declaring, 148
 - export lists in, 149–150
 - import lists in, 149
 - initialization order of, 150
 - See also* Modules
- Logical expressions, 8, 43–44
- Logical operators, 8, 43–44
 - short-circuit evaluation of, 8, 43–44, 44–45
- LONGCARD** (standard type), 37
- LONGINT** (standard type), 37
- LONGREAL** (standard type), 37
- LONGWORD** (standard type), 159
- Loop
 - body, 56
 - with exit in the middle, 58–59
 - infinite, 58
 - with multiple exit points, 59
- Looping, statements for, 49
- LOOP** statements, 9, 58–59
- Low-level features, 17–18, 153–166
 - See also* Absolute addresses; **ADDRESS**; **ADR**; **BIT-SET**; **BYTE**; **LONGWORD**; **SHORTADDR**; **SYSTEM**; Type transfer; **WORD**
- Make command, 145
- Mathematical functions, 8, 38–39
- MATHLIB** (library module), 14, 38–39, 60
- MAX** (standard function), 34
- Membership, set, 110
- MIN** (standard function), 34
- Mixed expressions, 37–38
- Mod** (**MATHLIB** function), 38
- MOD** (operator), 34
- Modula-2
 - applications of, 2
 - compared with Pascal, *see* Pascal
- Module priorities, 19, 28, 177–178
- Modules, 11–12, 129–151
 - advantages of, 129–130
 - compiling, 130
 - initialization order of, 139–140, 150
 - linking, 145
 - See also* Definition modules; Implementation modules; Library modules; Local modules; Program modules
- Modulus, 34
- Multidimensional arrays, 93–94
- Multiplication, 34, 36
- Multiplication operators, 44
- NEWPROCESS** (**SYSTEM** procedure), 18, 169–170
- NIL** (standard constant), 117–119
- Nodes, in a linked structure, 120
- Nonprinting characters, 40
- NOT** (logical operator), 8, 43
- NULLPROC** (standard constant), 125, 126
- Numbers, 23–24
- Numeric types, 33–39
 - See also* **CARDINAL**; **INTEGER**; **LONGCARD**; **LONGINT**; **LONGREAL**; **REAL**; **SHORTCARD**; **SHORTINT**
- Object files, 145
- Octal numbers, 4, 23

- ODD (standard function), 35, 37
 - One-pass compilers, 69–70
 - Opaque export, 140
 - Opaque types, 14–15, 133, 140–144
 - operations on variables of, 140
 - restricted to pointer types, 140–141
 - Open (FIO function), 157
 - Open array parameters, 10, 71, 95–96
 - Operating systems, 169
 - Operator associativity, 43, 44
 - Operator precedence, 34, 43, 44–45
 - Operators
 - arithmetic, 8
 - concatenation, 25
 - INTEGER, 34–35
 - logical, 8, 43–44
 - REAL, 36–37
 - relational, 8, 42–43
 - set, 8, 109
 - shift, 8, 25, 166
 - unary, 35
 - OR (logical operator), 43
 - ORD (standard function), 37–38, 40, 82–83
 - Order of compilation, 145
 - Order of module initialization, 139–140, 150
 - Order of procedure declarations, 69–70
 - Ordinal number
 - of a character, 39, 40
 - of an enumeration value, 82
 - Own variables, 68
 - Packing, 7
 - Parameters, 10, 65–67, 70–73
 - actual, 10, 67, 70–73
 - formal, 66, 70–73
 - open array, 10, 71, 95–96
 - procedures used as, 126
 - value, 10, 71, 72–73
 - variable, 10, 71, 72–73
 - Pascal, 1–11
 - array types, 6
 - Boolean expressions, 8
 - case-sensitivity, lack of, 4
 - case statements, 9
 - comments, 5
 - compilation, 11
 - compound statements, 9
 - conformant array parameters, 10
 - declarations, 5
 - enumeration types, 6
 - file types, 11
 - for statements, 4, 9
 - functions, 10
 - goto statements, 9
 - if statements, 9
 - input/output, 10–11
 - labels, 5
 - numbers, 4
 - operators, 8
 - packed types, 7
 - parameters, 10
 - pointer types, 7
 - procedural parameters, 7
 - procedures, 10
 - record types, 7
 - reserved words, 4
 - set types, 7
 - standard identifiers, 4
 - standard types, 6
 - statements, 9
 - string types, 7
 - subrange types, 6
 - type checking, 5
 - while statements, 9
 - with statements, 9
 - write, 4
 - writeln, 4, 10
 - Pointer constructors, 114

Pointer operations

- assignment, 117–118
- comparison, 119
- dereferencing, 116–118

Pointers, 7, 113, 114–123

- based, 114, 160
- dangling, 119–120
- returned by functions, 118–119

Pointer types, 7, 77–78, 113, 114–123

- as opaque types, 140–141
- specification of, 114

Pointer variables, 114–120**POW (MATHLIB function), 38****Precedence, operator, 34, 43, 44–45****Priorities, module, 19, 28, 177–178****Priority, interrupt, 178****PROC (standard type), 124****Procedure body, 66****Procedure declarations, 65–70**

- order of, 69–70

Procedure definitions, 134**Procedure heading, 66****Procedures, 10, 65–75**

- calling, 67
- function, *see* Functions
- proper, 65, 73
- recursive, 69–70
- returned by functions, 126–127
- standard, *see* Standard procedures
- used as parameters, 126

Procedure types, 7, 77–78, 113, 123–127

- PROC, 124
- specification of, 124

Procedure variables, 7, 123–126

- assignment to, 124–125
- calling, 125–126
- comparison of, 124

Processes, 169**Program modules, 11, 27–28**

- compilation of, 145

See also Modules**Proper procedures, 65, 73****Proper subset, 111****Pseudomodules, 158****Qualification, 28–30****Qualified export, 149–150****Qualified identifiers, 29****QUALIFIED option, in an export**

- list, 149–150

Qualifying identifiers, 29**Quotation marks, 24, 39, 97****Quotient, 34****Range checking, 92****RdKey (IO function), 41****REAL (standard type), 36–37****Real numbers, 24****REAL operators, 36–37****Recompilation, 145****Record aggregates, 100****Record fields, 99–101****Records, 7, 99–106**

- assignment of, 100
- with variant parts, 103–106

Record types, 7, 77–78, 99–106

- export of, 133
- specification of, 99

Recursion, 69–70, 74**Relational operators, 8, 42–43**

- applied to **BOOLEAN** values, 42
- applied to **CHAR** values, 40, 42
- applied to enumeration values, 82, 83
- for arrays, 93
- for pointers, 119
- for procedures, 124
- for records, 101
- for sets, 110–111

Relations, *see* Relational operators**Remainder, 34****REPEAT statements, 9, 57**

- Reserved words, 4, 22
- RETURN** statements, 9, 50
 - in functions, 10, 69, 73–74
 - in local modules, 150–151
 - in proper procedures, 68–69
- Right shift (operator), 8, 25, 166
- Same types, 86–87, 89
- Scale factor, 24
- Scientific notation, 24
- Scope, of field names, 99–100
- Selection
 - of an array component, 92, 94
 - of a record field, 100
- Selection statements, 49
- Selectors, 92
- Semicolon, 31–32
- Separate compilation, 15, 130, 144–146
- Set constants, 109
- Set elements, 108
- Set operators, 8, 109
 - difference, 109
 - equality, 110–111
 - inequality, 110–111
 - intersection, 109
 - membership, 110
 - subset, 110–111
 - superset, 110–111
 - symmetrical difference, 109
 - union, 109
- Sets, 7, 106–112
 - constructing, 108
 - empty, 106, 108
- Set types, 7, 77–78, 106–112
 - base type of, 107
 - BITSET**, 17, 107, 154–158
 - specification of, 107
- Set variables
 - adding elements to, 111–112
 - assignment to, 108
 - removing elements from, 111–112
- Shift operators, 8, 25, 166
- SHORTADDR** (standard type), 160
- SHORTCARD** (standard type), 37
- Short-circuit evaluation, of logical expressions, 8, 43–44, 44–45
- SHORTINT** (standard type), 37
- Side effects, 68
- Sin** (**MATHLIB** function), 38–39
- Single quote character, 24, 39, 97
- Sinh** (**MATHLIB** function), 38
- SIZE** (standard function), 115–116, 117
- Special symbols, 25
- Sqrt** (**MATHLIB** function), 38
- Standard constants
 - FALSE**, 42–43
 - NIL**, 117–119
 - NULLPROC**, 125, 126
 - TRUE**, 42–43
- Standard functions
 - ABS**, 35, 37
 - ADR**, 160
 - CAP**, 41
 - CHR**, 40
 - FLOAT**, 37–38
 - HIGH**, 95
 - MAX**, 34
 - MIN**, 34
 - ODD**, 35, 37
 - ORD**, 37–38, 40, 82–83
 - SIZE**, 115–116, 117
 - TRUNC**, 37–38
 - VAL**, 37–38, 82–83
- Standard identifiers, 4, 23
- Standard Pascal, *see* Pascal
- Standard procedures
 - DEC**, 51–52, 83–84
 - EXCL**, 111–112
 - HALT**, 9, 50
 - INC**, 17, 51–52, 83–84
 - INCL**, 111–112

Standard types, 6, 33, 77–78

See also ADDRESS; BITSET;

BOOLEAN; BYTE; CARDINAL; CHAR; INTEGER; LONGCARD; LONGINT; LONGREAL; LONGWORD; PROC; REAL; SHORTADDR; SHORTCARD; SHORTINT; WORD

Starting value, in a FOR statement, 61–63

Statements, 8–9, 30–31, 49–63

assignment, 8, 50–52

CASE, 9, 54–56

empty, 31

EXIT, 9, 50, 58–59

FOR, *see* FOR statements

GOTO, 5, 8–9, 50

IF, 9, 52–54

LOOP, 9, 58–59

procedure call, 67

REPEAT, 9, 57

RETURN, *see* RETURN statements

WHILE, 9, 56–57

WITH, 9, 102–103, 117

Statement separator, 32

Statement sequences, 30, 31

Statement terminator, 32

Static variables, 138, 151

Storage (library module), 115–116, 119

Str (library module), 99

String literals, 24–25, 39, 97–98

Strings, 6–7, 24–25, 39, 96–99

operations on, 25

termination of, 98

String types, 6–7, 96–99

specification of, 96–97

String variables, 97–98

assignment to, 97–98

Strong typing, 1, 5, 33, 86

Structured statements, 49

See also CASE statements;

FOR statements; IF statements; LOOP statements; REPEAT statements; WHILE statements; WITH statements

Structured types, 77–78, 91

See also Array types; Record types; Set types

Subrange types, 6, 77–78, 78–79, 84–86

base type of, 85

specification of, 85

Subroutines, *see* Procedures

Subset, 110–111

Subtraction, 34, 36

Superset, 110–111

Symbols, 4–5, 21–27

Symmetric difference, 8, 109

SYSTEM (pseudomodule), 17, 158–161

Tag field, 103–106

Tan (MATHLIB function), 38–39

TanH (MATHLIB function), 38

Terminating character, in a string, 98

Tilde, 8, 43

Transfer, coroutine, 167–168, 171–172

TRANSFER (SYSTEM procedure), 18, 171–172

Transparent export, 133

TRUE (standard constant), 42–43

TRUNC (standard function), 37–38

Type checking, 5

across module boundaries, 130, 133, 144–145

Type conversion, 37–38, 162–163

Type declarations, 79–80

Type definitions, 133

Type equivalence, 86–87, 89

Types, 5-7, 33, 77-80

ADDRESS, 17, 160
 anonymous, 79-80, 87
 array, 6-7, 77-78, 91-96
 basic, 33-47
BITSET, 17, 107, 154-158
BOOLEAN, 42-44, 81
BYTE, 159
CARDINAL, 3, 35-36
CHAR, 39-42
 enumeration, 6, 77-78, 78-79, 80-84
INTEGER, 34-35
LONGCARD, 37
LONGINT, 37
LONGREAL, 37
LONGWORD, 159
 numeric, 33-39
 opaque, 14-15, 133, 140-144
 pointer, 7, 77-78, 113, 114-123
PROC, 124
 procedure, 7, 77-78, 113, 123-127
REAL, 36-37
 record, 77-78, 99-106
 set, 7, 77-78, 106-112
SHORTADDR, 160
SHORTCARD, 37
SHORTINT, 37
 standard, *see* Standard types
 string, 6-7, 96-99
 structured, 77-78, 91
 subrange, 6, 77-78, 78-79, 84-86
 unstructured, 77-78
WORD, 17, 158-159
 Type specifications, 78-79
 Type transfer, 17-18, 161-164
 Unary operators, 35, 37
 Union, set, 109
 Unstructured types, 77-78

See also Enumeration types;
 Subrange types

VAL (standard function), 37-38, 82-83
 Value parameters, 10, 71, 72-73
 Variable declarations, 47, 79
 Variable definitions, 134
 Variable parameters, 10, 71, 72-73
 Variables
 anonymous, 7, 114-120
 lifetime of, 68, 138, 151
 locating at an absolute address, 18
 own, 68
 static, 138, 151
 Variant part, of a record, 103-106
 Variants, 103-106
 active, 104-106
 empty, 106
 Version checking, 145-146
 Vertical bar
 in a **CASE** statement, 55-56
 in the variant part of a record, 105-106

WHILE statements, 9, 56-57
 Whole-array operations, 93, 94, 96
 Whole-record operations, 100-101
 Wirth, Niklaus, 1
WITH statements, 9, 102-103, 117
WORD (standard type), 17, 158-159
WordMove (**Lib** procedure), 161
 Words, 158-159
 Workspaces, coroutine, 169-170, 174-175
WrCard (**IO** procedure), 4
WrChar (**IO** procedure), 41
WrLn (**IO** procedure), 4
WrReal (**IO** procedure), 47

TechKitTM

The TopSpeed Modula-2 TechKit offers an abundance of useful technical information, utilities, and program modules.

Write communications programs.

Asynchronous communications are made easy by the RS module supplied in the TechKit. RS is an interrupt-driven communications driver written entirely in Modula-2. It features easy installation and buffered I/O. Full source supplied together with a demonstration communications program.

Write assembler routines using TopSpeed Assembler.

TopSpeed Asm assembles 30,000 lines/minute and has been designed especially for TopSpeed Modula-2. Unlike other assemblers, TopSpeed Asm generates library files directly and is, therefore, superbly suited for use with the TopSpeed smart linker. TechKit contains complete documentation on the TopSpeed Assembler.

Write pop-up Terminate and Stay Resident programs a la Sidekick.

Using the module TSR supplied with Techkit, it is a breeze to write a pop-up application. Simply place all of your program in a single procedure and pass the procedure to TSR together with your memory requirement and how you want your program to be activated.

Implement a critical error handler.

DOS interrupt 24H is the critical error handler that is called when a hardware error occurs (such as an open disk drive door, a printer that is off-line, etc.). The DOS version of this interrupt handler usually displays: Abort, Retry, Ignore? With the TechKit's critical error handler, you can avoid this by taking control of this interrupt.

Use TopSpeed Modula-2 for imbedded systems.

TechKit contains all the necessary information to make ROMable code. You receive a ROMable version of the run-time library in TopSpeed Assembler, ready to modify for your specific requirements. Also, you will receive a program that enables you to specify exactly where the various segments of your program are to be placed.

Make use of Lotus/Intel/Microsoft expanded memory.

TechKit contains a complete set of procedures to interface to the Expanded Memory Manager (EMM).

Complete assembler source for start-up code.

To give you complete control of your program's final code, the TechKit contains the start-up code and run-time code, which are added to all programs automatically.

To order, mail in the coupon on the next page, or call us 24 hours, 7 days a week:

In the US: 1-800-443-0100 Ext 255
In England dial toll free: 0800-444-134



Jensen & Partners UK Ltd.
63 Clerkenwell Road
London EC1M 5NP
England

Jensen & Partners International, Inc.
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
USA