Modula-2 Manual

Robert P. Cook

9/1/94

1.0 A Comparison of Pascal and Modula-2

Modula-2[1,2] grew out of a practical need for a general, efficiently implementable, systems programming language. Its ancestors are Pascal[3] and Modula[4]. From the latter, it has inherited the name, the important module concept, and a systematic, modern syntax; from Pascal, most of the rest. This includes in particular the data structures, i.e. arrays, records, variant records, sets, and pointers. Structured statements include the familiar IF, CASE, REPEAT, WHILE, FOR, and WITH statements.

This Report reviews the differences between Pascal and Modula-2. It is not intended to teach you how to program in Modula-2. For that purpose, the definitions by Wirth[1, 2] should be consulted.

1.1 Identifiers

Identifiers are defined the same as in Pascal. Modula-2, however, is case sensitive. For example, the keyword "IF" is only recognized in its all caps form.

Examples:

x scan starMod firstLetter test1

1.2 Numbers

The Pascal number format is expanded to allow octal and hexadecimal constants to be expressed. Furthermore, the type CARDINAL is added to explicitly represent unsigned, 16-bit integers, and LONGINT is provided for 32-bit integers. Some important values for these types are as follows:

	MIN( )	**MAX( )**
INTEGER	-32768	32768
CARDINAL	0	65535
LONGINT	-2147483648D	2147483647D
REAL	-1.0E-35	1.0E+35

A decimal LONGINT constant is different from an INTEGER in that it must have a "D" following the last digit. Even a very large number must have the D. For an octal or hexadecimal LONGINT, the value of the number must either be too large for an integer or it must have enough leading zeroes to make the number at least six digits long.

Examples:

1980	decimal
3764B	octal (denoted by the trailing "B")
0CADH	hexadecimal (denoted by the trailing "H")
CADH	an identifier, not a number
48H	hexadecimal, leading zero is not required
236713D	decimal LONGINT (denoted by the trailing "D")
356165B	octal LONGINT
000121B	octal LONGINT with leading zeroes
36FA51H	hexadecimal LONGINT
000029H	hexadecimal LONGINT with leading zeroes

REAL numbers are supported in Modula-2 in precisely the same manner as Pascal. All REAL numbers must have a decimal point and must start with a digit; although, digits are not required in the fraction. An exponent field is also supported, but is optional. Following the fraction portion of the REAL number, an "E" must precede the exponent. The exponent has a range from -35 to 35. The unary plus ("+") can be placed on positive exponents as an option.

Examples:

5.32	typical REAL
433.	REAL without fraction portion
3.34E-22	REAL with negative exponent
83.28E31	REAL with positive exponent
12.3E+22	also a positive exponent

1.3 Characters and strings

Both the double quote character (") and single quote (') may be used as quote marks. However, the opening and closing marks must be the same character, and this character cannot occur within the string. A string must not extend over the end of a line. A string, consisting of a single-character, is of type CHAR; a string consisting of n>1 characters is of type ARRAY [0..n-1] OF CHAR.

By convention, many of the library modules use the null character, ASCII code 0, to delimit the end of a string. The storage for constant strings ends with the null automatically. Any string that the user creates should end with the null in order to work properly with string functions.

There is also a notation to represent characters that are not in the language's character set. A sequence of digits terminated with a "C" is interpreted as an octal value of type CHAR. For example, "123C" has the same value and type as "CHAR(123B)".

Examples:

"." '.' 123C "Modula"

"Don't Worry!" 'a "quoted" word'

1.4 Operators, delimiters, and comments

Operators and delimiters are the special characters, character pairs, or reserved words listed below. The reserved words consist exclusively of capital letters and MUST NOT be used as identifiers. The Modula-2 symbols, which differ from Pascal's, are listed separately.

Symbols That Are The Same As Pascal

+	-	*	/	:=	.	,	;	(	[	{	}	]	)
^	=	<	>	<>	<=	>=	..	:	AND		ARRAY		
BEGIN	CASE		CONST		DIV		DO		ELSE				
END	FALSE	FOR	FORWARD		IF		IN		MOD				
NIL	NOT	OF	OR		PROCEDURE				RECORD				
REPEAT	SET		THEN		TO		TRUE		TYPE				
UNTIL	VAR		WHILE		WITH								

Symbols Deleted From Pascal

downto	replaced with a BY clause.
file	I/O was deleted from Pascal in favor of services provided by I/O modules.
function	PROCEDURE is used instead.
goto, label	replaced by the LOOP statement.
packed	the only choice in Modula-2.
program	replaced by MODULE.

Symbols In Modula-2 But Not Pascal

|	#	~	
BY	DEFINITION	ELSIF	
EXIT	EXPORT	FROM	IMPLEMENTATION
IMPORT	LOOP	MODULE	POINTER
PROC	QUALIFIED	RETURN	

Comments may be inserted between any two symbols in a program. A comment is an arbitrary character sequence opened by the bracket "(*" and closed by "*)". Comments may be nested and they do not affect the meaning of a program. The nesting allows arbitrary sections of a program to be commented out for testing purposes.

1.5 Declarations

As in Pascal, every identifier must be declared within a block. A block in Modula-2, however, can be delimited by either the MODULE or PROCEDURE keyword. Unlike Pascal, the declarations within a block can occur in any order and can be repeated. Another difference is that constant expressions can be used wherever a constant is allowed. Finally, since this implementation is a one-pass compiler, ALL SYMBOLS MUST BE DECLARED BEFORE USE.

1.5.1 Constant declarations

Constant declarations are the same as Pascal, except for the use of constant expressions.

Examples:

```
CONST N     =     100;              (* N stands for 100 *)
   LIMIT    =     2*N-1;            (* LIMIT is for 199 *)
   ODDS     =     BITSET{1, 3, 5}
```

1.5.2 Type declarations

The simple types in Modula-2 consist of enumeration types, subrange types, or type identifiers, which may be qualified. In this context, the term "qualified" means preceded by a module identifier and a period. This option is not present in Pascal. The qualification may be necessary to refer to a type that is in a QUALIFIED EXPORT list or the definition module of another module. The following simple types are denoted by standard identifiers:

INTEGER	A variable of type INTEGER assumes as values the integers between MIN (INTEGER) and MAX (INTEGER).
CARDINAL	A variable of type CARDINAL assumes as values the integers between 0 and MAX (CARDINAL).
BOOLEAN	A variable of this type assumes the truth values TRUE or FALSE. These are the only values of this type, which is predeclared as the enumeration, BOOLEAN=(FALSE,TRUE).
CHAR	A variable of this type assumes as values the characters of the ASCII character set.
BITSET	A variable of this type assumes as values any subset of the SET OF [0 .. WordSize-1].
LONGINT	A variable of this types assumes the integer values between MIN (LONGINT) and MAX (LONGINT).
REAL	This type of variable can hold the fractional expressions between MIN (REAL) and MAX (REAL).
PROC	This type is a parameterless procedure.

The type of the bounds for a subrange type, T, is called the base type of the subrange and all operators applicable to operands of type T are also applicable to variables declared with the subrange type name. However, a value to be assigned to a variable of a subrange type must lie within the specified interval. If the lower bound is a non-negative integer, the base type of the subrange is taken to be CARDINAL; if it is a negative integer, it is INTEGER. The only difference from Pascal with respect to enumeration and subrange types is the requirement that a subrange declaration be bracketed.

Examples:

```
TYPE NewInt = INTEGER;
  Color  = (RED, BLUE, GREEN);
  Cold   = [-463 .. 58];              (* no brackets in Pascal *)
  Pnew   = POINTER TO ModuleName.New;
                                      (* a qualified reference *)
  Range  = [BLUE..GREEN];             (* a subrange of Color *)
  Letter = ["a" .. "z"];              (* the letters "a" to "z" *)
```

Modula-2 handles type equivalence much more strictly than Pascal. In Pascal, it is perfectly legal to assign variables of two different types as long as the two types "look" alike. Two types look alike if the component parts of the two declarations match exactly. With Modula-2, two separate types cannot be assigned to each other no matter how closely their declarations match.

Example:

```
VAR
    a : ARRAY [0..2] OF INTEGER;
    b : ARRAY [0..2] OF INTEGER;

a := b;  NO!   This is allowed in Pascal, but in
               Modula-2, a and b are variables of
               two different types.
```

1.5.2.1 ARRAY, SET, and POINTER types

The array and pointer types are interpreted and referenced as in Pascal. The array declaration is a bit different in that the bounds list is defined as a list of simple type names, enumerations, or subranges. The pointer declaration is more verbose than in Pascal. The purpose is to make the declaration "stand out" as the "^", used in Pascal, is easily overlooked. As in Pascal, NIL is used to specify an unbound pointer.

One of the exceptions to the "declare before use" rule concerns pointer types. In the case "POINTER TO T", T is automatically treated as a forward reference if it has not already been defined.

Examples:

```
TYPE Demo  =
  ARRAY CHAR, (RED, BLUE, GREEN) OF CHAR;
  Array = ARRAY [1 .. 9], [12 .. 347] OF CARDINAL;
  pChar = POINTER TO CHAR;
  pLinks = POINTER TO Links;          (* forward reference *)
  Links = ARRAY [1..4] OF pLinks;     (* defined *)

VAR  x : Demo;   (* referenced with x['j', BLUE] *)
```

Sets are declared as in Pascal but the syntax for a reference to a set constant is different. "{" and "}" are used to bracket set constants, whereas Pascal uses "[" and "]". The element designators can be constants or expressions. Sets are also restricted in size to WordSize elements. This must be a subrange of the integers between 0 and WordSize-1, or a subrange of an enumeration type with at most WordSize values. As a final point, a set constant may be preceded by a type name to document the interpretation of the element list.

Examples:

```
TYPE sColor = SET OF Color;
     BITSET = SET OF [0 .. WordSize-1];
```

Set Constants

{}	the empty set constant
{BLUE, RED}	the union of two colors
sColor{BLUE}	a set consisting of one color
BITSET{0..4, 6}	includes bits 0, 1, 2, 3, 4, 6

1.5.2.2 Record types

The syntax for the Modula-2 record type is similar to the Pascal notation, except for the format of the variant parts. In Pascal, the variant list is parenthesized. In Modula-2, the variant part is implemented as CASE selection. Each sub-declaration (case) in a variant part is delimited by a "|". Also, an ELSE option is provided to denote "all other cases". Another difference is that variant declarations can occur anywhere in a record type declaration, whereas in Pascal, variants are restricted to the end of a record declaration.

Example:

```
TYPE Ex = RECORD
     x,y : BITSET;
     CASE tag0 :Color OF       (* tag0 selects the case *)
          RED, GREEN: a,b : CHAR
        | BLUE:     c: INTEGER
                                (* "|" separates variant parts *)
END (* Ex *);
```

The example contains two variant sections. The case within the first variant is selected by the value of "tag0", the case within the second variant by "tag1". Remember that, as in Pascal, the variant parts of each case overlay each other in storage.

1.5.2.3 Procedure types

Unlike Pascal, Modula-2 permits variables of procedure type that can have procedure names as values. This feature can be useful when the function to be performed is to be selected at runtime. Since the procedure type is generic, that is, it stands for an arbitrary number of procedure names, the identifiers in the formal parameter list are omitted; only the type names appear. For procedure variables without a formal parameter list, the type PROC may be used.

Examples:

```
TYPE
     prMax  = PROCEDURE(INTEGER, INTEGER)
                        : INTEGER;
     prSecToDate =PROCEDURE(VAR Seconds) : Date;
     parLess = PROC;
```

Procedure variables are initialized by the assignment of either other procedure variables or procedure constants, which result from procedure declarations.

1.5.3 Variable declarations

Variable declarations serve to introduce variables and associate each with a unique identifier and a fixed data type. Variables whose identifiers appear in the same list all obtain the same type.

Examples:

```
VAR i,j : CARDINAL;
    a   : ARRAY Index OF CHAR;
```

1.5.4 Procedure declarations

Procedure declarations consist of a procedure heading and a block that is called the procedure body. The heading specifies the procedure identifier and the formal parameters. The block contains declarations and statements. The procedure identifier is required at the end of a procedure declaration to document which procedure is being "closed". The primary differences from Pascal are procedure variables, the deletion of the "function" keyword, and the addition of the RETURN statement. Rather than assigning to the procedure identifier to set a return value as in Pascal, a RETURN statement must be used.

```
PROCEDURE identifier [FormalParameters] ";"
  {Const | Type | Var | Procedure | Module Declaration}
[BEGIN
  StatementSequence]
END identifier

FormalParameters =
 "(" [FPSection { ";" FPSection}] ")" [":" qualifiedIdent]

FPSection =
    [VAR] identifierList ":" [ARRAY OF] qualifiedIdent

qualifiedIdent = identifier { "." identifier}
```

The use of a FORWARD qualifier in place of a procedure body allows a procedure to be referenced before its declaration. The FORWARD immediately follows the procedure heading. When the actual procedure is declared, however, the full formal parameter list must be repeated.

Example:

```
PROCEDURE foo (x : CARDINAL);
FORWARD;              (* replaces body *)

PROCEDURE fip;
BEGIN
     foo (14);        (* use before declaration *)
END fip;

PROCEDURE foo (x : CARDINAL);
BEGIN
    InOut.WriteCard (x,4);
END foo;
```

1.5.4.1 Formal parameters

Formal parameters are identifiers that denote actual parameters specified in the procedure call. As in Pascal, both value and variable (VAR) parameters are provided. Formal parameters are local to the procedure, i.e. their scope of reference is the program text that constitutes the procedure declaration.

Example:

```
(* Read a string of digits from the input device. *)
(* The Cardinal value of the digits is returned. *)
(* Conversion starts when a digit is read. *)
(* Conversion stops when a non-digit is read. *)
PROCEDURE ReadCard() : CARDINAL;
     VAR i : CARDINAL; ch : CHAR;
BEGIN
     REPEAT          (* skip characters until a digit is read *)
          InOut.Read(ch);
     UNTIL (ch>="0") AND (ch<="9");
      i := 0;
     REPEAT          (* accumulate the number in "i" *)
          i := 10*i+(ORD(ch)-ORD("0"));
          InOut.Read(ch);
     UNTIL (ch<"0") OR (ch>"9");
     RETURN i;
END ReadCard;
```

The "ReadCard" routine uses the type transfer function, ORD, to manipulate the numeric value of the input character.

Any function with an empty parameter list, such as "ReadCard", must be declared and referenced with the "()" suffix. The goal is to create a visual distinction between a reference to a procedure variable and a procedure call.

The specification of "open" array parameters represents a significant improvement over the static limitations of Pascal. If the parameter is an "open" array, the form

ARRAY OF Type

must be used, where the specification of the actual index bounds is omitted. "Type" must be compatible with the element type of the actual array, and the index ranges are mapped onto the integers 0 to N-1, where N is the number of elements. If the initial array is multidimensional, it is mapped onto the argument with the last subrange listed first. That is if the array's index bounds is defined as [0..2,0..2], the argument will be mapped [0,0]->[0], [0,1]->[1], [0,2]->[2], [1,0]->[3], etc. The "HIGH" standard function can be used to determine "N-1". The example illustrates the use of this feature in an error message routine.

```
PROCEDURE error(VAR message :ARRAY OF CHAR);
          (* Notice: the bound for "message" is omitted *)
     VAR nChar : CARDINAL;
BEGIN
     WriteLn;                    (* skip to new line *)
     FOR nChar := 0 TO HIGH(message)DO
                                 (* no. chars in message *)
          Write(message[nChar]);  (* write the message *)
     END; (*for*)
     WriteLn;                    (* skip to new line *)
END error;

error("short");                  error("MEDIUM1");
                 error("longest one");
```

The "open" array feature also makes it easy to create libraries of useful routines that can operate over a wide range of input values.

1.5.4.2 Standard procedures

The standard procedures are as follows:

ABS(x)	absolute value; result Type=argType
CAP(ch)	capitalize ch
CHR(x)	the character with ordinal number x
FLOAT(x)	converts x to a REAL value
HIGH(x)	the upper bound of array x
MIN(x)	the minimum value for type x
MAX(x)	the maximum value for type x
ODD(x)	x MOD 2 <> 0
ORD(x)	ordinal number of x in its enumeration
SIZE(x)	the number of words in type x
TRUNC(x)	the LONGINT value of a REAL or the INTEGER value of a LONGINT
LONG(x)	the LONGINT value of an INTEGER or CARDINAL x.
VAL(T, x)	is the value with ordinal number x and type T VAL(T, ORD(x))=x, if x is of type T
DEC(x);	x := x-1;
DEC(x, n);	x := x-n;
EXCL(s, i);	s := s-{i}; remove i from set s
HALT;	terminate program execution
INC(x);	x := x+1;
INC(x, n);	x := x+n;
INCL(s, i);	s := s+{i}; include element i in s

Examples:

ABS(-5) = 5	ODD(3) = TRUE
CHR(65) = 'A'	ORD('A') = 65
CAP('a')= 'A'	VAL(Color, 0) = RED

```
     x:=8;    y:={0,4,5};
```

```
DEC(x); x = 7          DEC(x, 5); x = 3
INC(x); x = 9          INC(x, 5); x = 13
EXCL(y, 4);            y = {0,5}
INCL(y, 6);            y = {0,4,5,6}
```

1.5.4.3 Conversion and Type transfer functions

Conversion functions perform the useful service of converting one number type into another by actually changing the argument's bit values. FLOAT takes an INTEGER, CARDINAL, or LONGINT value and converts it to REAL; FLOAT's inverse, TRUNC, takes a REAL argument and converts it into LONGINT. TRUNC also provides the more docile but no less important role of converting LONGINT values into INTEGER, which involves the removal of the high-order bits.

The other conversion functions perform similar bit additions or removals. LONG takes an INTEGER or CARDINAL value and makes it LONGINT. CHR removes the high byte of an INTEGER or CARDINAL value to make it an ASCII value of type CHAR. ORD, the inverse of CHR, adds a high byte of zeroes back on to create a CARDINAL.

Type transfer functions are different from conversion functions in that they do not change any bits. Type transfer functions merely convert the argument into a new type at compile time. Of course, the new type must have the exact size as the old. ORD, for example, performs a dual role; it is the conversion function mentioned above, and it also gives the ordinal value of its argument in the argument's enumeration. VAL is the inverse of this. It takes the enumeration's type name and its ordinal value and makes them into the enumeration's type. The other way to transfer types is to use the type name as a function. Again, the two types must be of equal size. Type transfer between CARDINAL and INTEGER is automatic on assignment.

Examples:

```
TYPE
    Arr = ARRAY [0..3] OF CARDINAL;
    Rec = RECORD
        m : LONGINT;
        n : LONGINT;
    END;
VAR
    c : CARDINAL;
    i: INTEGER;
    l : LONGINT;
    ch : CHAR;
    r : REAL;
    a : Arr;
    r : Rec;

r := FLOAT(42); (* r = 42.0 *)
l := TRUNC(r);          (* l = 42D *)
c := TRUNC(l);          (* c = 42  *)
l := LONG(c)            (* l = 42D *)
i := TRUNC(l);          (* i = 42  *)
i := ORD('A');          (* i = 65  *)
ch := CHR(i);           (* ch = 'A' *)
c := 14;                (* c = 14  *)
i := c;                 (* i = 14  *)
c := l;                 (* c = 14  *)
a := Arr(r);            (* r is made into the array *)
```

1.6 Expressions

The following table defines the interpretation of each operator.

Operator	Meaning
+	integer addition
-	integer subtraction
*	integer multiplication
DIV	integer division
MOD	integer modulus

OR
p OR q means "if p then TRUE, otherwise q"

AND &
p & q means "if p then q, otherwise FALSE"

NOT ~
~ p means "if p then FALSE, otherwise TRUE"

=	compare for equality
<> #	unequal
<	less
<=	less than or equal
>	greater
>=	greater than or equal

IN	contained in, set membership test
+	x IN (s1 + s2) iff (x IN s1) OR (x IN s2)
-	x IN (s1 - s2) iff (x IN s1) & ~ (x IN s2)
*	x IN (s1 * s2) iff (x IN s1) & (x IN s2)
/	x IN (s1 / s2) iff (x IN s1) <> (x IN s2)
<=	p <= q is TRUE if p is a proper subset of q
>=	p >= q is TRUE if q is a proper subset of p

Examples:

```
3+4 = 7             3-4 = -1
7 DIV 4 = 1         3*4 = 12
7 MOD 4 = 3         TRUE OR FALSE = TRUE
TRUE AND FALSE = FALSE
NOT TRUE = FALSE
3 = 4 is FALSE    3 <> 4 = TRUE
3 < 4 = TRUE      3 <= 4 is TRUE
3 > 4 = FALSE   5 >= 4 is TRUE
5 IN {4,5} = TRUE          {4,5} + {4,7} = {4,5,7}
{4,5} - {4,7} = {5}        {4,5} * {4,7} = {4}
{4,5} / {4,7} = {5,7}      {4,5} <= {4,5,7} = TRUE
{4,5,7} >= {4,5} = TRUE
```

1.7 Statements

The major difference in statement structure from Pascal involves the elimination of the distinction between simple and compound statements. In other words, "BEGIN S {; S} END" has been deleted by making every structured statement a compound statement. REPEAT, for example, was already in this form and required no change. The advantage of the new format is that statements can be arbitrarily added

without worrying about whether a "BEGIN-END" is necessary. To facilitate this property, we recommend that every statement be terminated with a semicolon. Except for the compound statement convention, the following statements are similar to the syntax used in Pascal. The WITH statement is restricted to a single record selector.

ForStatement =

```
    FOR identifier ":=" expression TO expression [BY ConstExpression] DO
        StatementSequence
    END (* FOR *)
```

RepeatStatement =

```
    REPEAT
        StatementSequence
    UNTIL expression
```

WhileStatement =

```
    WHILE expression DO
        StatementSequence
    END (* WHILE *)
```

WithStatement =

```
    WITH recordReference DO
        StatementSequence
    END (* WITH *)
```

The Modula-2, FOR loop uses the optional BY clause to specify the step value to be used in each iteration. The step must be a constant. If the step is positive, the loop counts up to the TO value. If the step is negative, the loop counts down to the TO value. The following rules should be obeyed when using FOR loops:

- The bounds expressions should not depend on anything in the body of a loop.
- The control variable should not be modified within the loop.
- The value of the control variable should be considered undefined after loop termination.
- The control variable can not be an imported variable, PROCEDURE parameter, RECORD member or array element.

Examples:

```
FOR i := 3 TO 7 DO              i=3,4,5,6,7
FOR i := 3 TO 7 BY 2 DO         i=3,5,7
FOR i := 7 TO 1 BY -2 DO        i=7,5,3,1
```

1.7.1 Assignments and type compatibility

The assignment serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator is written ":=" and is pronounced as becomes.

assignment =

```
    variableReference ":=" expression
```

The type of the variable must be assignment compatible with the type of the expression. Operands are said to be assignment compatible, if either they are compatible, or both are of type INTEGER or CARDINAL or subranges with base types INTEGER or CARDINAL. Two operands of types T0 and T1 are compatible if either T1 = T0, or T1 is a subrange of T0, or T0 is a subrange of T1, or if T0 and T1 are overlapping subranges of the same base type. In the case of overlapping subranges, runtime checks for range violations may be necessary to detect errors.

1.7.2 CASE statement

The CASE statement in Modula-2 is somewhat different than the Pascal version. First, subrange constants are allowed as a shorthand notation for a range of case labels.

Pascal	Modula-2
3,4,5,6,7 :	3..7 :

The subrange notation saves typing. Furthermore, constant expressions can also be used as case labels. Thus, defined constants can be used to parameterize selection. Finally, the "|" is used to separate cases and an ELSE clause is adopted as a shorthand for the label standing for all other labels. No value may occur more than once as a case label. The maximum number of cases per case statement is 256.

CaseStatement =

```
    CASE expression OF
    Case
    {"|"Case}
    [ELSE
        StatementSequence]
    END (* CASE *)
```

Case =

```
    [CaseLabels {"," CaseLabels} ":"
        StatementSequence]
```

CaseLabels =

```
    ConstExpression [".." ConstExpression]
```

Example:

```
(* Read a string of digits from the input device. *)
(* "," and "." are allowed in the string for readability. *)
(* The Cardinal value of the digits is returned. *)
(* Conversion starts when a digit is read. *)
(* Conversion stops when a non-digit is read. *)
PROCEDURE ReadCard() : CARDINAL;
    VAR i : CARDINAL;
        ch : CHAR;
BEGIN
    REPEAT     (* skip characters until a digit is read *)
        InOut.Read(ch);
    UNTIL (ch>="0") AND (ch<="9");
    i := 0;
    LOOP                        (* accumulate the number in "i" *)
        CASE ch OF
        "0".."9":  i := 10*i+(ORD(ch)-ORD("0"));
        | ",", ".":             (* ignore "," and "." *)
        ELSE                    (* stop at non-digit *)
            EXIT; (* loop *)
        END; (* case *)
        InOut.Read(ch);
    END; (* loop *)
    RETURN i;
END ReadCard;
```

1.7.3 IF statement

The IF statement has been modified by the addition of an ELSIF clause whose purpose is to provide a shorthand notation for tests that, in Pascal, would require multiple IF statements.

```
IfStatement =
    IF expression THEN
        StatementSequence
    {ELSIF expression THEN      (*zero or more *)
        StatementSequence}
    [ELSE                       (*zero or one ELSE *)
        StatementSequence]
    END (* IF *)
```

Example:

Pascal	**Modula-2**
if x = 1 then	IF x = 1 THEN
y := 2	y := 2;
else if x = 9 then	ELSIF x = 9 THEN
y := 3	y := 3;
else	ELSE
y := 6;	y := 6;
	END; (* IF *)

The expressions following the symbols IF and ELSIF are of type BOOLEAN. They are evaluated in the sequence of their occurrence until one yields the value TRUE. Then, the associated statement sequence is executed and the IF terminated. If an ELSE clause is present, it is executed if and only if all Boolean expressions yielded the value FALSE, much like the ELSE in the CASE construct.

1.7.4 LOOP and EXIT statements

A loop statement specifies the continuous execution of a statement sequence. This statement is used quite frequently in concurrent algorithms because, unlike sequential programs, termination is often undesirable. Imagine what would happen if an operating system halted after 10,000 iterations.

```
LoopStatement =
    LOOP
        StatementSequence
    END (* LOOP *)

ExitStatement = EXIT
```

Example:

```
TYPE pList = POINTER TO List;
    List = RECORD
        link : pList;           (* a singly-linked list *)
        attribute : Attribute;  (* a list element      *)
    END; (* List *)

PROCEDURE search(list : pList;
                 VAR attribute : Attribute) : BOOLEAN;
    (* check to see if "attribute" is in "list" *)
BEGIN
    LOOP  (* search singly-linked list *)
        IF list = NIL THEN
            EXIT;
        ELSIF attribute = list^.attribute THEN
            RETURN TRUE; (* attribute is in the list *)
        END; (* IF *)
        list := list^.link;    (* advance to next element *)
    END; (* LOOP *)
    RETURN FALSE;          (* end of list; not there *)
END search;
```

The EXIT statement specifies termination of the loop and, when executed, causes execution to continue at the statement following the loop statement. An EXIT statement may terminate a LOOP even if it is nested within other structured statements. Only the closest, enclosing LOOP is terminated.

1.7.5 RETURN statement

The RETURN statement provides a convenient way to leave a procedure as soon as an exit condition becomes true. In Pascal, a procedure can only be terminated by executing the "end" of the block, which is often an inconvenience.

```
RETURN [expression]
```

In Modula-2, the RETURN statement serves the dual role of specifying the result for a function and of returning to the caller for both subroutines and functions. For a subroutine, the expression must be omitted. For a function, it must be present. The expression, representing the returned value, must match the type specified for a function.

2.0 Programming Conventions

In addition to the indentation conventions used in the Modula-2 definition, you should try to, and we will, adhere to the following programming conventions. Hopefully, the result will be visually pleasing programs that are easier to understand due to the presence of syntactic cues.

2.1 Names and declarations

Declarations should help document the use of a variable; thus, try to use subrange and enumerated type declarations instead of INTEGER. Most identifiers should be written in lower case, except for the first letter of each new word, that should be capitalized.

```
line    firstLine   nextLineOffset
```

Capitalize the first letter of type identifiers, module names, and the names of exported procedures; capitalize all letters of CONST definitions. If the name of a constant is several words, just capitalize the first two letters of the first word(e.g. CHarsPerWord). Try to use full words for all names. However, if space is a problem, the following shorthand conventions can be used.

Choose a short tag for each basic type that you create, e.g. Ln for Line or Buf for Buffer. Use the following prefixes to construct tags for derived types:

```
p - pointer to:         pBuf = POINTER TO Buf
i - index for:          iLn  = index for ARRAY OF Ln
s - set of:             sColor=SET OF Color
sr- subrange of:        srColor=[BLUE..GREEN]
n - length of:          nString=number of characters in
```

If you need only one variable of a given type in a scope, use the tag as its name:

```
buf : Buf
```

If you need several names, append modifiers (avoid simple numbers like 1, 2, etc.):

```
bufOld, bufNew, bufAlt : Buf
```

2.2 Layout

Try to follow the indentation examples in the Modula-2 definition. Write one statement per line, unless several simple statements, which together perform a single function, will fit on one line. It is acceptable to put a loop on a single line if it will fit. If a statement will not fit on a single line, indent the continuation line(s).

A semicolon follows the last statement in a statement sequence and the last field in a field list. The purpose is to make insertions and deletions less error-prone.

Each DEFINITION module should be commented to describe its general function. Also, each exported procedure should have a brief comment. In addition, it is advisable to comment VAR parameters as "IN", "OUT", or "INOUT" to denote the presence or absence of side-effects.

2.3 Spaces

Leave a space after a comma or semicolon and none before; leave a space before and after a colon. Surround ":=" with spaces. A space should appear after left-comment and before right comment. Don't put spaces inside brackets or parentheses or around single-character operations.

3.0 Changes to Modula-2

The following list reflects a number of changes to the Modula-2 definition[5]. The changes resulted from a meeting between Wirth and representatives of several firms that had implemented Modula-2.

1. All objects declared in a definition module are exported. The explicit export list is discarded. The definition module may be regarded as the implementation module's separated and extended export list.

```
DEFINITION MODULE identifier ";"
    {import}
    {definition}
END identifier "."
```

2. The syntax of a variant record type declaration is changed so that the ":" is always required. The presence of the colon makes it evident which part was omitted, if any.

```
CASE [identifier] ":" qualifiedIdent  OF
```

3. The syntax of the case statement and the variant record declaration is changed so that either may be empty. The inclusion of the empty case and empty variant allows the insertion of superfluous bars similar to the insertion of superfluous semicolons for empty statements.

4. A string consisting of N characters is said to have length N. A string of length 1 is compatible with the type CHAR.

5. The syntax of the subrange type is changed to allow the specification of an identifier designating the base type of the subrange. Example: INTEGER[0 .. 99].

6. The syntax of sets is changed to allow expressions as set element selectors.

```
set = [qualifiedIdent] "{" [element {"," element}] "}"
element = expression [".." expression]
```

7. The character "~" is a synonym for NOT.

8. The identifiers LONGCARD, LONGINT, and LONGREAL denote standard types (which may not be available on some implementations).

9. The type ADDRESS is compatible with all pointer types and with either LONGCARD or LONGINT depending on the implementation.

10. The new standard functions MIN and MAX take as an argument any scalar type, including REAL. They stand for the type's minimal/maximal value.

REFERENCES

[1] Wirth, N., Modula-2. Technical Report No. 36, Institut fur Informatik der ETH Zurich, (Dec. 1980).

[2] Wirth, N., Programming in Modula-2. Springer-Verlag New York Inc., (1982).

[3] Wirth, N. and K. Jensen., Pascal user manual and report. Springer-Verlag New York Inc., (1976).

[4] Wirth, N., Modula: a language for modular multiprogramming**. Software--Practice and Experience 7**, (1977), 3-35.

[5] Wirth, N., Schemes for multiprogramming and their implementation in Modula-2. Technical Report No. 59, Institut fur Informatik der ETH Zurich, (June 1984).

AN INTRODUCTION TO MODULAR PROGRAMMING

1.0 Introduction

Modula-2 was designed to support modular programming. This section outlines the features of Modula-2 which reflect that goal. Also, the facilities of Modula-2 for systems programming are illustrated.

Many systems today are large programs, ranging in size from ten thousand to one-half million lines of code. Obviously, some design guidelines are necessary to manage the complexity of implementing and maintaining such large systems. The most successful approach has been to use modular programming techniques[1] that allow one module to be written with little knowledge of the implementation of other modules and that allow modules to be recompiled and replaced without requiring recompilation of an entire system. The expected benefits of modular programming are shortened development time for new products because modules can be implemented by separate groups, increased flexibility because the implementation of one module can be changed without the need to change others, and increased comprehensibility because the system can be studied one module at a time.

In system design, the first step is to partition the specification into a number of modules with well-defined interfaces. At this point, only the interfaces are considered, not the module implementations. Each module should be small and simple enough to be thoroughly understood and well programmed. The intention is to describe all "system level" decisions (i.e. decisions that affect more than one module). The modularization must take into account both the functions to be provided to users, resulting in top-down decisions, and the technological constraints imposed by the possible execution environments, resulting in bottom-up decisions.

In choosing a modularization for a system, it is advantageous to impose a hierarchical organization on the modules. A hierarchical structure results when all modules at level i in a system use only modules at levels lower than i for their implementation. A module at level 0 is implemented without referring to any other modules. The existence of a hierarchical structure assures us that upper levels can be deleted and arbitrarily rebuilt. This property enhances the extensibility, or "open"ess, of a system. If "low-level" modules were implemented such that they depended upon "high-level" modules, a hierarchy would not exist and it would be much more difficult to delete or update portions of the system.

2.0 Modular Programming

The following listing illustrates the syntax of a compilation unit in Modula-2.

Modula-2 Program Structure

```
CompilationUnit = DefinitionModule | [IMPLEMENTATION] ProgramModule

ProgramModule =
      MODULE identifier";"
            {import}
      block identifier "."


DefinitionModule =
      DEFINITION MODULE identifier ";"
            {import}
      {definition}
      END identifier "."

 import =
      [FROM identifier] IMPORT IdentifierList ";"


export =
       EXPORT [QUALIFIED] IdentifierList ";"


definition =
      CONST {ConstantDeclaration ";"}  |
      TYPE  {identifier ["=" type] ";"} |
      VAR   {VariableDeclaration ";"}  |
      PROCEDURE identifier [FormalParameters] ";"
```

A program module encapsulates the implementation of an abstraction. A compiler, for example, might have modules for symbol table lookup, reading from the input stream, accumulating tokens, and generating code.

To meet our modularity requirements, a module must be easily recognized. In addition, its function should be easy to determine. This does not mean examining the listing of the entire module. In fact, for proprietary software, the module listing may not be available. As you will learn in this section, Modula-2 meets, and exceeds, all of our requirements. We start with an example of a Modula-2 program module that prints the integers between one and a hundred, and their squares.

PRINT THE SQUARES OF THE INTEGERS 1..100

```
MODULE Main;

      FROM InOut IMPORT       (* Procedures *)
            WriteCard, WriteString, WriteLn;
      VAR i : [1..100];

BEGIN
      WriteString("Number  Number Squared");
      WriteLn;
      FOR i := 1 TO 100 DO
            WriteCard(i, 4);                  (* aligns number under "b" *)
            WriteCard(i*i, 16);               (* aligns under "S" *)
            WriteLn;                          (* writes end-of-line      *)
      END; (* for *)
END Main.
```

The major difference between the Modula-2 version of the program and its Pascal equivalent is the IMPORT list and the variety of I/O procedures. Modula-2 has no builtin I/O statements; therefore, all I/O is performed with procedures written in Modula-2. This design decision resulted in a simpler implementation for the compiler but increased typing for users.

The IMPORT list is necessary to tell the compiler where to find the definitions for the "Write" procedures, in this case in module "InOut", which has been separately compiled. The IMPORT list also enumerates the symbols from "InOut" that are required by the "Main" program.

"InOut" is an example of a low-level module that can be used over and over again by high-level modules. In fact, program modules, such as "Main", must always occur at the highest system level as they can only "import" definitions from lower-level modules like "InOut".

If a global variable is not listed in the IMPORT list, it is invisible to the module. Thus, by examining the interface specification at the top of a program module, a user can determine what services the module depends on from its environment (useful documentation). Since the modularization process starts by defining module interfaces, the IMPORT list is usually determined prior to implementation. Any symbol that is used by a module and does not appear in the IMPORT list must be declared in the body of the module.

If the FROM clause in an IMPORT list is omitted, the list of identifiers must name modules, not symbols contained in modules. In this case, all of the symbols occurring in the DEFINITION part of the named modules are made available to the program. However, these symbols can only be referenced via a qualified name of the form ModuleId.SymbolId. The following example illustrates the qualified name option.

PRINT THE SQUARES OF THE INTEGERS 1..100

```
MODULE Main;
      IMPORT InOut;    (* only the module name *)
      VAR i : [1..100];
BEGIN
      InOut.WriteString("Number  Number Squared");
      InOut.WriteLn;
      FOR i := 1 TO 100 DO
            InOut.WriteCard(i, 4);          (* aligns number under "b" *)
            InOut.WriteCard(i*i, 16);       (* aligns under "S"*)
            InOut.WriteLn;                  (* writes end-of-line      *)
      END; (* for *)
END Main.
```

2.1 DEFINITION modules

Modula-2 permits the definition specification for a module to be separated from the module's implementation. The two parts can be compiled separately but must, of course, match with respect to declarations. A DEFINITION module supports information hiding by eliminating the implementation code. It is intended to be standalone documentation for the users of an abstraction. Furthermore, in most Modula-2 implementations, the IMPLEMENTATION part can be recompiled arbitrarily without causing additional recompilations on the part of its users. If a DEFINITION module is recompiled, all modules that refer to it must be recompiled.

A DEFINITION module contains only the constant, type, variable, and procedure-heading declarations that are necessary to use the corresponding IMPLEMENTATION module. The interface specification lists the entities that are "export"ed to the outside world by the module and any entities from the outside world that are "import"ed (used) by the DEFINITION module. The following example illustrates a portion of the "InOut" DEFINITION module. Notice that only the procedure headings are given. The procedure bodies are specified in the IMPLEMENTATION module for "InOut".

THE InOut DEFINITION MODULE

```
(* Provides formatted I/O services for basic types *)
DEFINITION MODULE InOut;

PROCEDURE WriteCard(x, n : CARDINAL);
(* write cardinal x with (at least) n characters.
   If n is greater than the number of digits needed,
   blanks are added preceding the number. *)
PROCEDURE WriteLn(); (* terminate the current line *)
PROCEDURE Write(ch:CHAR);
(* write a single character *)
PROCEDURE WriteString(s : ARRAY OF CHAR);
(* write HIGH(s)+1 characters from s *)

END InOut.
```

The full details of types exported from DEFINITION modules are visible to importing modules. If an enumeration or record type is exported, the enumerated constant and field names are automatically exported as well. This is termed a transparent export.

At the other extreme, it is possible to export only a type's name. This is referred to as opaque export. The term "opaque" denotes the hiding of the details of a type's implementation from its users. An opaque type is declared as follows:

An Opaque Type Declaration

```
TYPE identifier;
```

In the corresponding IMPLEMENTATION module, an opaque type can only be declared as a pointer or a simple type, such as CARDINAL. Instances of opaque types can be used only for assignment, comparison, or as arguments to procedures defined in the corresponding IMPLEMENTATION module.

2.2 IMPLEMENTATION modules

A correctly structured module has the property that its implementation can be changed without changing the parts of the program outside the module. This property by itself would suffice as a reason to use Modula-2 over Pascal.

It is important to document the external symbols that are used in an IMPLEMENTATION module. Notice that the IMPORT list for the DEFINITION and IMPLEMENTATION parts need not match. Typically, the IMPLEMENTATION module's list will be longer as greater detail is necessary to implement an abstraction as opposed to specifying it.

Every IMPLEMENTATION module contains an initialization part, following the "BEGIN", that is used to put the module into a consistent state before program execution starts. The initialization code is executed by the runtime system before the main program begins. Therefore, it is unwise to put infinite loops in an initialization part.

The next example illustrates the use of a DEFINITION and IMPLEMENTATION module to define a stack manipulation utility. The program implements a single stack that has its size and its element's type chosen by its users. In the example, "stack" and "iStack" are not exported because they are implementation details. By "hiding" them, the programmer responsible for maintaining the module can continue to refine and improve its implementation without affecting any of its users. For instance, the stack could be implemented as a linked list rather than an array.

In addition to serving as a convenient organizational tool, the module also provides an information-hiding and parameterization service. The user of the module can call "Push", "SetEmpty" and "Pop", but all implementation details are hidden. In the example, the module imports the type of the stack's elements and the size of the stack. Thus, this module could be used to create the following varieties of stacks:

Possible Content of the "Parameters" Module

```
CONST MAxStackSize = 42;
TYPE StackType = INTEGER; (*a stack of 42 integers*)

CONST MAxStackSize = 97;
TYPE StackType =BOOLEAN;
      (*a stack of 97 Booleans*)
```

The advantage of this parameterization is that the stack module takes on a life of its own, independent of any particular program. Any algorithm that needs a stack can "check out" this module from a system library, read its specification, set up the parameters, and not worry about coding it. Notice that, unlike

procedure parameters, the imported type and constant are evaluated and have their effect only at compile time.

A Stack Manipulation Example

```
(* This module implements a single stack together with the
   operators that manipulate it.  To use this module, create
   a Parameters module that defines MAxStackSize, which
   is the number of elements desired, and StackType. *)
DEFINITION MODULE StackManipulation;

FROM Parameters IMPORT
     (*Type*) StackType; (* restricted to a simple type *)

PROCEDURE Push(stackElement : StackType):BOOLEAN;
(* adds to top; returns FALSE if a push doesn't succeed *)
PROCEDURE Pop(VAR stackElement : StackType):BOOLEAN;
(* removes from top; returns FALSE if stack was empty *)
PROCEDURE SetEmpty();
(* sets the stack to empty *)

END StackManipulation.


IMPLEMENTATION MODULE StackManipulation;

(* ********INTERFACE SPECIFICATION******** *)
FROM Parameters IMPORT
     (*Const*) MAxStackSize, (*Type*) StackType;
(* ************DECLARATIONS************** *)
VAR
     stack : ARRAY [1 .. MAxStackSize] OF StackType;
     iStack : [1 .. MAxStackSize+1];

(* **********IMPLEMENTATION PART******** *)
PROCEDURE Push(stackElement : StackType)
                         :BOOLEAN;
BEGIN
     IF iStack <= MAxStackSize THEN
          stack[iStack] := stackElement;
          INC(iStack);                    (* the same as iStack:=iStack+1 *)
          RETURN TRUE;
     ELSE
          RETURN FALSE;                   (* error-stack overflow *)
     END; (* if *)
END Push;

PROCEDURE Pop(VAR stackElement : StackType):BOOLEAN;
BEGIN
     IF iStack > 1 THEN
          DEC(iStack);     (* the same as iStack:=iStack-1 *)
          stackElement := stack[iStack];
               (* exit with a value *)
          RETURN TRUE;
     ELSE
          RETURN FALSE;       (* error-stack underflow *)
     END; (* if *)
END Pop;

PROCEDURE SetEmpty();
BEGIN
     iStack := 1;
END SetEmpty;

(* **********INITIALIZATION PART********** *)
BEGIN
     SetEmpty();
END StackManipulation.
```

2.3 Module-based abstractions

In this Section, we review some of the more common techniques for implementing a data abstraction. System designers must choose among these methods when designing the user interfaces. The previous StackManipulation example illustrates one of the choices. Notice that it is restricted to implementing exactly one stack per use of the module. The other data abstraction choices are to export a type, to export an opaque type, and to export an index. The StackManipulation module is used as an example for each method.

2.3.1 Exported type

The first choice to implement an abstraction is to export a type, such as "StackOfIntegers". The advantage of this approach is that the new abstraction extends the language available to the programmer. The new type can be used to declare variables in the same way as any builtin type like INTEGER or CHAR. Instances of these variables are then passed as arguments to the StackManipulation procedures.

The disadvantage of the approach is that the implementation details of the type are visible and accessible to the users. As a result, a change in representation requires a recompilation by all users of the module and may invalidate some programs. Thus, this design choice should be used with extreme care for any user interface provided by an operating system. Another disadvantage is the inability to share at runtime a single StackManipulation module for stacks of different type.

A Stack Manipulation Example With An Exported Type

```
DEFINITION MODULE StackManipulation;

FROM Parameters IMPORT
     (* Const *) MAxStackSize,  (*number of stack elements*)
     (*Type*) StackType;         (* the element type *)

(* This module implements a stack type together with the
   operators that manipulate it.  To use this module, create
   a Parameters module that defines MAxStackSize, which
   is the number of elements desired, and StackType, which
  can be of any type.  *)
TYPE Stack = RECORD
     iStack: [1..MAxStackSize+1];
     stack : ARRAY [1..MAxStackSize] OF StackType;
END; (*Stack*)

PROCEDURE Push(VAR stack : Stack; VAR element :StackType):BOOLEAN;
(* adds to top; returns FALSE if a push doesn't succeed *)
PROCEDURE Pop(VAR stack : Stack; VAR element :StackType):BOOLEAN;
(* pops from top to "element"; returns FALSE if stack was empty *)
PROCEDURE SetEmpty(VAR stack : Stack);
(* sets a stack to empty *)

END StackManipulation.
```

2.3.2 Opaque type

The second technique uses an opaque type, a pointer, to represent the stack abstraction. When the user declares instances of the Stack type, only uninitialized pointers are allocated. Thus, the implementation must provide a "NewStack" operator to allocate a stack of a particular size and a "FreeStack" operator to deallocate stacks.

A Stack Manipulation Example With An Opaque Type

```
DEFINITION:

TYPE Stack;
PROCEDURE NewStack(VAR stack : Stack;
          stackSize : CARDINAL) :BOOLEAN;
(* allocate stack;
  return FALSE on storage allocation error *)
PROCEDURE FreeStack(VAR stack:Stack):BOOLEAN;
(* deallocate stack;
   return FALSE on storage allocation error *)

IMPLEMENTATION:

TYPE Stack = POINTER TO StackDescriptor;
StackDescriptor = RECORD
     allocated : BOOLEAN;(* set to TRUE by NewStack *)
     size : CARDINAL;              (* set from stackSize *)
     iStack : [1..MAxStackSize+1];
     pStack : POINTER TO ARRAY [1..MAxStackSize]
                    OF StackType;
END; (* StackDescriptor *)

PROCEDURE NewStack(VAR stack : Stack; stackSize :CARDINAL) : BOOLEAN;
BEGIN
     IF (stackSize=0) OR (stackSize>MAxStackSize) THEN
          RETURN FALSE;
     END;
     Storage.ALLOCATE(stack, TSIZE(StackDescriptor));
     IF stack = NIL THEN
          RETURN FALSE;
     END;
     stack^.allocated := TRUE;
     stack^.size := stackSize;
     stack^.iStack := 1;
     Storage.ALLOCATE(stack^.pStack, stackSize);
     IF stack^.pStack = NIL THEN
          Storage.DEALLOCATE(stack,
                    TSIZE (StackDescriptor));
          RETURN FALSE;
     END; (* if *)
     RETURN TRUE;
END NewStack;
```

The advantage of this approach is the ability to bind the size of a stack at runtime. In other words, the IMPLEMENTATION module must allocate the space for each new stack. The disadvantage is again the inability to define a "class" of stacks that would allow the component type to be specified arbitrarily.

2.3.3 Index

The last option uses the same DEFINITION module as the previous example. But in this case, the opaque type is declared as a CARDINAL rather than a pointer. The IMPLEMENTATION module maintains an array of pointers to StackDescriptors. The array index, which is used as the argument to the module's procedures, selects a descriptor from the array. The pointer from the descriptor is then used to manipulate a stack, just as was done with the previous example. The array simply represents an additional level of indirection. The advantage of the index technique is that it supports validity checking. That is, it is easy to determine if a given index is really associated with a stack. Validity checking is more difficult when using pointers since there is no way to force a user to initialize instances of the "Stack" type.

3.0 Low-Level Programming Facilities

In order to implement some systems in Modula-2, it must be possible to deal with machine dependencies and it must be possible to bypass the compiler's type checking. We discuss the latter requirement first. (These low-level operations should be used carefully and only when absolutely necessary.

3.1 Eliminating type checking

The first facility to breach Modula-2's type checking is type transfer functions. A type identifier can be used as a function to transfer a parameter to the type identifier's type. In most implementations, no conversion is performed; type transfers have their effect at compile time.

Type Transfer Examples

```
CHAR(65)      =   'A'
CARDINAL('A')  =    65
BITSET(3)+BITSET(5) = 7
```

3.2 The SYSTEM module

The second set of capabilities is provided by module SYSTEM, which is "builtin" to the compiler. The definition of SYSTEM is implementation dependent.

Low-Level SYSTEM Facilities

```
DEFINITION MODULE SYSTEM;
(* IMPLEMENTATION DEPENDENT *)

TYPE
     ADDRESS=POINTER TO WORD;
     (*assignment compatible with pointer types*)
     WORD;            (* compatible with any simple type *)
PROCEDURE ADR(x : (***ANY TYPE***) ): ADDRESS;
(* turns any variable reference into an ADDRESS type. *)
PROCEDURE TSIZE(x : (*ANY TYPE IDENTIFIER*) ): CARDINAL;
(* returns the number of address units that "x" occupies.
```

It operates on a type's name, not on instances of the type. *)

END SYSTEM.

The SIZE (builtin) and TSIZE functions allow the implementor to obtain machine specific information. For example, the size of an integer array big enough to store a 512-word disk sector can be obtained with the expression "512 DIV TSIZE(INTEGER)". Since the size of a word in our implementation is one machine unit, TSIZE(INTEGER) returns the value one. The use of these functions improves the portability of an operating system.

The ADDRESS and WORD types support the implementation of generic routines, particularly for I/O. Both types bypass the compiler's type checking. Modula-2 also supports the convention that if a formal parameter is specified as ARRAY OF WORD, then any variable, structured or unstructured, can be supplied as an argument. The ADR function can be used to initialize a pointer to the address of any data structure. As an example, the following routine takes an arbitrary array of characters and prints it in slices of "unit" characters at a time.

Print Slices of Strings

```
PROCEDURE printSlice(VAR s:ARRAY OF WORD;
                     size, width:CARDINAL);
VAR
    i,j:CARDINAL;
    c:POINTER TO ARRAY [0..9999] OF CHAR;
BEGIN
    j := 0;
    c := ADR(s);              (* use a pointer to access *)
    FOR i := 0 TO size-1 DO (* byte-wise for each    *)
        InOut.Write(c^[i]);    (* char in the argument *)
        INC(j);
        IF j >= width THEN
            (* print "width" characters *)
            InOut.WriteLn;   (* then start a new line  *)
            j := 0;
        END; (* if *)
    END; (* for *)
    IF j <> 0 THEN
        InOut.WriteLn;
    END;   (* end line, if necessary *)
END printSlice;

Examples:

a := '0123456789';
printSlice(a, 10, 5);      prints      01234   56789
printSlice(a, 10, 3);      prints      012   345   678   9
```

3.3 Coroutines

The final low-level facility that is discussed is the notion of a coroutine. Wirth uses this abstraction to build higher-level operating system routines to manipulate a program; for example, to assign a program the CPU or to remove it from control of the CPU. The coroutine operators are fundamental to any operating system.

In a subroutine program structure, there is a master/slave relationship between a calling program and its subroutine. Usually, a subroutine has one entry point and all local variables, except the formal parameters, are undefined at entry time.

Coroutines, on the other hand, are programs that may call each other, but do not have a master/slave relationship. On exit from a coroutine, its state (i.e. program counter, stack pointer) is saved in a variable of type Coroutine; the next time the coroutine is called, it resumes execution at exactly the point where it previously paused. All local variables and parameters retain their previous values.

The Coroutine type and the operators to manipulate coroutines are defined in the COROUTINE module, which again is machine dependent.

In Modula-2, a coroutine is created by specifying a procedure, which represents the actions of the coroutine, and a stack to hold the procedure activation records, which represent the execution state of the procedure. Before a coroutine can be "resumed" for the first time (e.g. start execution), its state must be initialized by calling InitCoroutine. The arguments to InitCoroutine are a procedure as well as a stack base address and size. The stack size must be chosen in an application-dependent way; in fact, some architectures do not even require this information.

The Transfer procedure implements the "resume" operation by saving the execution state of the current coroutine in a variable of type Coroutine and restoring the execution state of a second coroutine. A RETURN operation from a coroutine procedure is normally an error.

The COROUTINE Module

```
DEFINITION MODULE COROUTINE;
(* Routines to turn procedures into coroutines and to
   transfer control of the CPU from one coroutine to
   another. *)

TYPE
    Coroutine = POINTER TO RECORD   (* stores state of a coroutine *)
            pc : ADDRESS;  (*bare machine's program counter *)
            sp : ADDRESS;  (*bare machine's stack pointer *)
        (* ANY OTHER DATA NEEDED TO EXECUTE A Coroutine*)
        END RECORD;

PROCEDURE InitCoroutine(p:PROC; stack:ADDRESS;
            stackSize:CARDINAL;
                    VAR (* OUT *)corouine:Coroutine);
(* Initializes a coroutine record for procedure "p" so
    that a "Transfer" to "p" will start it executing. *)
PROCEDURE Transfer(VAR from, to : Coroutine);
(* Saves the hardware registers of the executing
    procedure in "from" and then resets the registers to
    the values in "to", resulting in a transfer of control. *)

END COROUTINE.
```

The following example uses three coroutines to illustrate the concepts. The first coroutine, "getChar", is used as a filter to reduce all sequences of three identical characters to the letter "J". Thus, "abbbbabbddd" as input would result in "aJbabbJ" as output. The second coroutine, "print", "resumes" the first to retrieve and print filtered characters. Since the "Main" program is initialized with a stack, it is automatically a coroutine.

When the "getChar" routine pauses, it leaves the filtered character in "resultChar". The program stops when it reads a ".", followed by any different character. Notice that the values of "ch" and "previousChar" in "getChar" are saved across Transfer operations.

A Coroutine Example

```
MODULE Main;
IMPORT InOut, COROUTINE;
VAR
    startCo, getCo, printCo : COROUTINE.Coroutine;
    stack1, stack2 : ARRAY [1..200] OF INTEGER;  (* stack space *)
    resultChar : CHAR;

PROCEDURE print();
BEGIN
    REPEAT
        COROUTINE.Transfer(printCo, getCo);  (* resume "getChar" coroutine *)
        InOut.Write(resultChar);
    UNTIL resultChar = ".";      (* stop on ".*" sequence *)
    COROUTINE.Transfer(printCo, startCo); (* resume "main" program *)
END print;

PROCEDURE getChar();
VAR ch, previousChar : CHAR;  (* these values are preserved *)
BEGIN
    InOut.Read(previousChar);
    LOOP
        InOut.Read(resultChar);
        IF previousChar = resultChar THEN  (* do two in a row match? *)
            InOut.Read(resultChar);
            IF previousChar = resultChar THEN  (* do three in a row match? *)
                InOut.Read(previousChar);
                resultChar := "J";
            ELSE
                ch := resultChar;  (* no, return two, then proceed *)
                resultChar := previousChar;
                COROUTINE.Transfer(getCo, printCo); (* resume "print"  *)
                resultChar := previousChar; (* falls through to Transfer *)
                previousChar := ch;
            END; (* if *)
        ELSE
            ch := previousChar;  (* two characters are different *)
            previousChar := resultChar;
            resultChar := ch;           (*  set return value  *)
        END; (* if *)
        COROUTINE.Transfer(getCo, printCo);  (* resume "print" coroutine *)
    END; (* loop *)
END getChar;

BEGIN
    COROUTINE.InitCoroutine(getChar, stack1, SIZE(stack1),  getCo);
    COROUTINE.InitCoroutine (print, stack2, SIZE(stack2), printCo);
    COROUTINE.Transfer(startCo, printCo); (* save "Main"; resume "print" *)
    InOut.WriteLn;
    InOut.WriteString("End Of Program");
END Main.
```

4.0 Compiling and Executing

The Modula-2 environment is composed of three units: two compilers and one runtime. The compilers perform all the needed code generation, and the runtime executes the code once it is selected.

4.1 The Definition Compiler

The first compiler is called "d" for definition compiler. The definition compiler is the precursor to the second compiler. Its job is to decipher definition modules to produce the implementation interface. Every file "d" receives must have the suffix ".def". If this suffix is not supplied, the compiler will add it automatically. The implementation's interface is stored in a file with the same filename as the source code except the suffix is changed to ".SBL". An implementation interface is required for each reference to an imported module. The compiler searches for any needed .SBL files in the current directory. If it does not find one, it prompts the user to input the path name that locates the needed file.

All definition modules must be compiled before they are used (imported). Once a definition module is compiled, it should not be compiled again unless it is extended. When a definition module is changed, first, compile all dependent definition modules and then secondly, compile all dependent implementation modules.

4.2 The Program Compiler

The second compiler is called "c" for compile. This program takes the ASCII file of a program or implementation module and forms it into the object code that the runtime uses. Files sent to the compiler require the ".mod" suffix in order for the compiler to recognize it as Modula-2 source code. If the suffix is omitted, the compiler will append it

automatically. Also when the suffix is omitted, any error messages generated during compliation will be immediately printed. Object files have the same name as the source text with the suffix switched to ".OBJ".

Error messages generated by either compiler can always be found in the file called filename.LST where filename is the name of the file that the compiler attempted to translate. When errors occur, the compiler will try to continue compilation beyond the error. This is so that all errors can be discovered before the user attempts to compile again. When any errors occur during compilation, neither object code nor implementation interfaces will be generated.

4.3 The Runtime

To execute any compiled program the runtime, "x", is called upon. Only program modules created by the program compiler can be run. The runtime can take no arguments; the filename must be supplied only when the runtime requests it. The runtime will ask whether the user wants to use the trace option. If the user responds "y" then the runtime will display each line of object code in hexadecimal and octal. Unless the user understands the internal object code, this option ought not be used. When the runtime asks for the filename two options can be used. Either the full name can be given, or the name can be given without the suffix following the period.

Example:

Consider the stack manipulation example as it was first given. A carriage return follows every command.

First, the Parameters definition module must be created:

```
DEFINITION MODULE Parameters;
```

```
CONST
        MAxStackSize = 10;
TYPE
        StackType = INTEGER;  (* something simple *)
END Parameters.
```

Now, Parameters can be compiled with the definition compiler.

```
Type in:
        d Parameters.def
        or
        d Parameters

The computer will respond with:

Parameters.def
Modula-2
in>
 + Parameters.SBL.
```

Everything is now prepared for StackManipulation's definition module to be compiled.

```
Type in:
        d StackManipulation.def
        or
        d StackManipulation

The computer will respond with:

StackManipulation.def
Modula-2
in>
 Parameters: Parameters.SBL

 + StackManipulation.SBL.
```

The last action the user must perform is compiling the implementation module. The program compiler is used for this.

```
Type in:
        c StackManipulation.mod
        or
        c StackManipulation

The computer will respond:

StackManipulation.mod
Modula-2
in>
 StackManipulation: StackManipulation.SBL

 Parameters: Parameters.SBL

 + StackManipulation.RFC....
 + StackManipulation.OBJ 113
```

The first names listed are all the files (e.g. StackManipulation, Parameters) the module wishes to import, which may include the module's own DEFINITION module. The last names listed are the new files the compiler created and the size of these files if appropriate.

StackManipulation is now ready to be used in any program the user creates. If the user wishes to change either StackType of MAxStackSize, Parameters must be edited and all three files must be compiled again.

Whenever a program requests the use of an imported module, the runtime must bring that module into memory. The first time the runtime encounters a reference to an imported module, the runtime will fetch its object code into memory and execute its initialization statements if it has any. This is called dynamic linking. The object code must be located in the current directory or else the runtime will not be able to find it. The initialization sequence is only performed once. If many modules import the same module, that module's initialization code will only be performed when it is first read in.

If the runtime encounters an error, a message will be printed. Since all runtime errors are fatal to the program, execution will immediately stop. These are all the possible error messages that the runtime can issue:

normal exit
HALT statement
CASE error
stack overflow
heap overflow
missing RETURN in a function
address error
REAL overflow
REAL underflow
bad operand
CARDINAL overflow
INTEGER overflow
subrange or subscript error
division by zero
illegal instruction
breakpoint

REFERENCES

[1] Parnas, D.L. On the criteria to be used in decomposing systems into modules**.** **Communications of the ACM 15**, 12 (Dec. 1972) 1053-1058.