WISCONSIN MODULA
Part III of the First Report on
THE CRYSTAL PROJECT

by

Raphael Finkel
Robert Cook
David DeWitt
Nancy Hall
Lawrence Landweber

Computer Sciences Technical Report # 501

April 1983

Wisconsin Modula

Part III of the First Report on The Crystal Project

Raphael Finkel

Robert Cook

David DeWitt

Nancy Hall

Lawrence Landweber

CONTENTS

1. Introduction to Crystal

The University of Wisconsin Crystal project was funded starting in 1981 by the National Science Foundation Experimental Computer Science Program to construct a multicomputer with a large number of substantial processing nodes. The original proposal called for the nodes to be interconnected using broadband, frequency-agile local network interfaces. Each node was to be a high performance 32 bit computer with a approximately 1 megabyte of memory and floating-point hardware. The total communications bandwidth was expected to be approximately 100 Mbits/second.

During the first year of the project, these specifications have been refined. We have decided to buy approximately 40 node machines, each a VAX-11/750. The interconnection hardware will be the Proteon ProNet. Currently, the ProNet is available in a 10 Mbits/second version. We have contracted with Proteon to increase the effective bandwidth to 80 MBits/second.

1.1. Software Overview

The purpose of this hardware is to promote research in distributed algorithms for a wide variety of applications. In order to provide different applications simultaneous access to the network hardware, we have designed a software package called the *nugget* that resides on each node. In brief, the nugget provides the following facilities:

1. The nugget enforces allocation of the network among different applications by virtualizing communications within partitions of the network. These partitions are established interactively through a host machine.
2. Backing store is shared among the nodes by nugget facilities to virtualize disks.
3. Interaction between the user and individual machines is provided by the nugget facility of virtual terminals.

4. Initial loading, control, and debugging of programs on node machines is controlled by nugget software.

The Charlotte operating system is designed to provide standard interactive operating system support within a Crystal partition. The Charlotte *kernel* provides

1. processes
2. multiprocessing
3. inter-process communication that hides node boundaries
4. mechanisms for scheduling, store allocation, and migration.

All policies in Charlotte are concentrated in *utility processes*. They are designed so that each such process controls a policy on its own set of machines. The set may range in size from one machine to the entire partition. The processes that control the same resource on different machine sets communicate with each other to achieve global policy decisions. The utilities that have been designed so far include a switchboard, a program starter, and the file server. In addition, there are non-policy utilities for command interpretation and program connection.

We expect that Crystal will be used for a wide range of applications. Currently research is underway in distributed operating systems, programming languages for distributed systems, tools for debugging distributed systems, multiprocessor database machines, parallel algorithms for math programming, numerical analysis and computer vision, and evaluating alternative protocols for high performance local network communications.

All Crystal software is being written in a local extension to Modula. Our compiler, which runs on a VAX running Berkeley 4.1 Unix, employs syntactic error correction through the FMQ algorithm and is quite fast. The code it generates compares well with that produced by the C compiler.

1.2. Phases of the project

The first phase of the project was dedicated to defining both the hardware and the software. This phase ended in December 1982. Decisions were reached concerning both the node machines and the interconnection devices. The node machine decision was difficult. We had to balance our concerns for reliability, availability, speed, and cost. The machine we chose, the VAX-11/750, although not as fast as others we investigated, had the advantage of being a known architecture for which our Modula compiler already generates code. The Proteon network is currently available. We have been using this network to interconnect our Unix VAX machines and have found it to be extremely reliable.

During the first phase, the nugget was specified and a prototype implementation was completed on a network of eight Digital Equipment PDP-11/23 computers connected by the Megalink CSMA broadband network manufactured by Computrol. Charlotte was also specified and the kernel debugged on this network.

The second phase of the project has just gotten underway. We are finalizing the nugget specifications, which changed in minor ways when we decided that the node machines would be VAXen. The nuggetmaster, which controls the partitions, has also been specified. Charlotte is undergoing debugging of the utility processes. During this phase, which lasts until July 1984, we will transfer the nugget and Charlotte to the node machines and modify them as necessary for the ProNet. Charlotte will be modified to fit with the nugget. (Until now, they have been developed independently.) The utility processes will be supplemented with login and authentication processes, and the file system will be converted to use Crystal disks instead of a file system on the host machine. We plan to have a production, stable operating system by the end of this phase.

The third phase of the project will see large-scale applications actively pursued. Some of this work will start during the second phase. We also expect to re-evaluate the hardware decisions at some point during this phase. There is some reason to expect that frequency-agile modems will be available that will make communication within each partition truly independent of communication within other partitions. Each partition will be able to use its own set of frequencies. Work with optical fiber technology for computer interconnection is also underway at various laboratories around the country. Within five years, impressive bandwidths should be available, reaching into the gigabit/second range. We will continue to monitor progress in this area.

1.3. This report

The purpose of this report is to describe the current state of the design and implementation of the Crystal project. It is intended for readers who have no familiarity with Crystal and wish to see the design decisions that have been made. It is also intended for implementers who need a coherent and reasonably complete specification in order to interface various parts of the project. This dual readership requires us to repeat ideas, first presenting them in an overview fashion, and then diving into tedious details. We urge the reader to skip over those parts of the document that are not at the right level of detail. This report is divided into several documents.

This document describes the Modula language and its local implementation at the University of Wisconsin. The language description is based heavily on "Modula: a Language for Modular Multiprogramming", *Software Practice and Experience 7*, pp. 3-35, 1977 by N. Wirth. Points of difference between the standard language and our implementation are noted throughout.

The implementation is in C and was written by Keith Thompson during the first half of 1982. It has been maintained and largely rewritten by Nancy Hall since July

1982. It can generate code for either the VAX or the PDP-11 and can be retargeted for other machines as well.

2. Differences from standard Modula

Our version of Modula provides:

separate compilation
constant expressions
qua (casting)
Interface module procedures may call any procedure,
not just standard ones.
Send operations may be applied to imported signals.
panicsig
value parameters
forward declarations
second argument to low, high
bits and <enum> type transfer functions
printf
Unix interface procedures
otherwise in case statements
process statements anywhere a statement is legal
various sizes for integers
floating point
variable list in with statements
compiler options inside meaningful comments

This version does not provide:

device modules

The syntax is different for:

initializations
process stack size
case statements
min, max (-maxint, maxint)
string literals

The following have been defined:

the maximum depth of procedure nesting
character set (Ascii)
compatibility of variables (weak name equivalence)

3. Notation for syntactic description

An extended Backus-Naur formalism is used to describe the syntax. It allows use of syntax expressions as right-hand parts in a production. Syntactic entities are denoted by English words expressing their intuitive meaning. Symbols of the language are either enclosed by quote marks (") or are in boldface and appear as literals in the right-hand parts of productions. Each production has the form

S = E.

where S is a syntactic entity and E a syntax expression denoting the set of sentential forms (sequences of symbols) for which S stands. An expression E has the form

$T_1 \mid T_2 \mid \ldots \mid T_n \; (n > 0)$

where the T_i's are the "terms" of E. Each T_i stands for a set of sentential forms, and '|' denotes their union. Each term T has the form

$F_1 \; F_2 \ldots F_n \; (n > 0)$

where the F_i's are the "factors" of T. Each F_i stands for a set of sentential forms, and T denotes their product. The product of two sets of sentences is the set of sentences consisting of all possible concatenations of a sentence from the first factor followed by a sentence from the second factor. Each factor F has either the form

"x"
reservedword

(x is a literal, and "x" denotes the singleton set consisting of this single symbol, or the literal is a reserved word), or

(E)

(denoting the expression E), or

[E]

(denoting the union of the set denoted by E and the empty sentence), or

{E}

(denoting the set consisting of the union of the empty sequence and the sets E, EE, EEE, and so on).

Examples:

The syntax expressions

("a" | "b") ("b" | "c")

"a" {"bc"}

"a" ["b" | "c"] "d"

denote the following sets of sentences respectively:

ab	ac	bb	bc
a	abc	abcbc	abcbcbc ...
ad	abd	acd	

4. Language vocabulary and representation

The language is a infinite set of sentences (programs), namely the sentences well formed according to the syntax. Each sentence (program) is a finite sequence of symbols from a finite vocabulary. The vocabulary consists of identifiers, (unsigned) numbers, literals, operators and delimiters. They are called lexical symbols or tokens, and in turn are composed of sequences of characters. Modula uses the Ascii character set. (Standard Modula does not require Ascii.)

Identifiers are sequences of letters and digits. The first character must be a letter. An underscore is considered a letter. For the PDP-11, identifiers should be distinguishable within their first 7 characters. (This restriction only applies to procedures and variables, and is not part of Standard Modula.)

ident = letter {letter | digit}.

Numbers (integers) are sequences of digits, possibly followed by the letter B (or b) signifying 'octal'. (Standard modula does not allow 'b'.) Floating point numbers have digits both before and after the decimal point. (Standard modula does not have floating point numbers.)

```
number = integer | float.
integer = digit {digit} | octaldigit {octaldigit} ("B" | "b").
float = integer "." integer.
```

Character literals are single characters enclosed in single-quote marks. The single character may itself be a single-quote mark. Special characters may be denoted in several ways. For example, the newline character, number 12 (octal) in Ascii, may be denoted as:

```
'\n'
12c
12C
'\12'
char(10);
```

The C or c notation is used to turn any integer (expressed in octal) into a character. (Standard Modula only allows 'C'.) The '\n' notation is useful for a few standard characters, like newline (n), carriage-return (r), and tab (t). The character '\<num>' is equivalent to <num>C; <num> is expressed in octal. (Standard Modula does not distinguish character literals from string literals, and does not allow the "char" or the "\" notation.)

```
character literal = octalnumber ("C" | "c") |
        "'\" (character | octalnumber) "'" |
        "'" character "'".
```

String literals are sequences of characters enclosed in double-quote marks. (Standard Modula uses single-quote marks.) If a double-quote mark itself is to occur within that sequence, then it is denoted by two consecutive double-quote marks.

```
string = """ {character} """.
```

The character string may contain characters in "\n" notation.

Operators and delimiters are special characters, character pairs, or reserved words listed below. In this report, they are in boldface for clear distinction from identifiers. These reserved words must not be used in the role of identifiers. (Standard Modula does not include some of these reserved words, and the Modula reserved word *device* is not reserved in this version.)

Operators and delimiters

```
+ - * / = . <> , < ; <= : > >= (* := *)
```

and	**array**	**begin**	**case**
const	**define**	**div**	**do**
else	**elsif**	**end**	**exit**
external	**forward**	**if**	**interface**
loop	**mod**	**module**	**not**
of	**or**	**otherwise**	**procedure**
process	**qua**	**record**	**repeat**
then	**type**	**until**	**use**
value	**var**	**when**	**while**
with	**xor**		

Blank spaces (and line separation) are ignored unless they are essential to separate two consecutive symbols. Hence, blanks cannot occur within symbols, including identifiers and numbers.

Comments may be inserted between any two symbols in a program. They are opened by the bracket (* and closed by *). Comments may be nested, and they do not affect the meaning of a program. However, *meaningful comments* can be used to affect the compiler state. For example, to turn listing on, one may insert a comment of this form:

```
(* $s+ *)
```

These comments have this form:

meaningfulcomment = "(* $" character ("+" | "-" | "=" number) " *)".

The character specifies the option (all options are described later). If "+" follows the

character, the option is turned on; "-" turns the option off. Numeric-valued options are set by "=". (Standard Modula does not have meaningful comments.)

5. Constant Declarations

A constant declaration associates an identifier with a constant value.

```
constantdeclaration = ident "=" constantexpression.
constantexpression = expression | constant.
constant = unsignedconstant | ("+" | "-") number.
unsignedconstant = ident | number | character | string | bitconstant.
bitconstant = "[" [bitlist] "]".
bitlist = bitlistelement {"," bitlistelement}.
bitlistelement = constant [":" constant].
```

A constant expression may be any expression all of whose subparts are either constants, constant expressions, or built-in functions of constant expressions. (Standard Modula does not allow constant expressions.)

Numbers are constants of type integer or float. The variant of integer or float chosen depends on the size of the constant (see "Basic types"). A constant denoted by a character literal (or constant character expression) is of type char (see "Basic types"). Constants of type char are not equivalent to constants of type string that happen to have length 1. A string consisting of n characters is of type "**array** 1 : n **of** char" (see "Array structures").

A bit constant is a constant of type bits (see "Basic types"). The elements of the bitlist are the indices of those bits that are "true". An element of the form m:n specifies that all bits with indices m through n are "true". All other bits have the value "false".

6. Type declarations

Every constant, variable and expression is of a certain type. In the case of numbers and literals their type is implicitly defined, for variables it is specified by their declaration, and for expressions it is derivable from the types of their

constituent operands and operators. A data type determines the set of values that a variable of that type may assume; it also defines the structure of a variable. There are five standard types, namely integer, float, Boolean, char and bits. (Standard Modula does not have float.) Enumeration types and the types integer, float, Boolean and char are unstructured, that is, their values are atomic. Structured types (structures) can be declared in terms of these elementary types and of structures.

```
typedeclaration = ident "=" type.
type = ident | enumeration | arraystructure | recordstructure.
```

6.1. Basic types

integer Values are whole numbers in the range -maxint to maxint, where "maxint" is a constant dependent on available implementations. For the PDP-11, maxint = 32767; for the VAX: maxint = 2147483647. (Standard Modula uses the constants "min" and "max".) There are several variants of integer that specify precision: shortint, midint, and longint. (Standard Modula does not have these variants.)

shortint This variant of integer occupies 8 bits. Its values are restricted to lie between -128 and 127.

midint This variant of integer occupies 16 bits. Its values are restricted to lie between -32768 and 32767.

longint This variant of integer occupies 32 bits. It is not available for the PDP-11. Its values are restricted to lie between -2147483648 and 2147483647.

float Values are real numbers in the range available on the target machine. Floats are not available on the PDP-11. Floats use 32 bits on the VAX. There are several variants of float that specify precision: ffloat and dfloat. (Standard Modula does not have floating

point numbers.)

ffloat This variant of float occupies 32 bits.

dfloat This variant of float occupies 64 bits.

Boolean Values are the truth values denoted by the predefined identifiers "true" and "false". The word "boolean" may be substituted for "Boolean" (but not in Standard Modula).

char Values are the characters belonging to the Ascii character set. (Standard Modula does not require Ascii.)

bits Values are arrays of w Boolean elements. This type is predefined as "**array** 0 : w **of** Boolean" (see "Array structures"). The constant w is the wordlength minus 1 of the computer on which Modula is implemented. (For the PDP-11, w=15; for the VAX, w=31.)

6.2. Enumerations

An enumeration is a list of identifiers that denote the values that constitute a data type. These identifiers are used as constants in a program. They, and no other values, belong to this type. An ordering relation is defined on these values by their sequence in the enumeration.

```
enumeration = "(" identlist ")".
identlist = ident {"," ident}.
```

6.3. Array structures

An array structure consists of a number of components that are all of the same component type. Each component is identified by a number of indices. This number is called the *dimensionality* of the array. The range of index values of each dimension is specified in the declaration of the array structure. The types of the indices must not be structured.

arraystructure = **array** indexrangelist **of** type.
indexrangelist = indexrange {"," indexrange}.
indexrange = constant ":" constant.

6.4. Record structures

A record structure consists of a number of components, called *record fields*. Each component is identified by a unique field identifier. Field identifiers are known only within the record structure definition and within field designators, that is, when they are preceded by a qualifying record variable identifier. The data type of each component is specified in the field list.

recordstructure = **record** fieldlist {";" fieldlist} **end**.
fieldlist = [identlist ":" type].

Examples of type declarations

```
color = (red, yellow, green, blue)
vector = array 1 : 100 of color
matrix = array 1 : 20, 1 : 10 of integer
account =
        record x : integer;
               y : Boolean;
               z : array 0 : 9 of char
        end
```

7. Variable Declarations

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

variabledeclaration = identlist ":" type.

Two variables are "compatible" if they have the same type, either because they are declared in the same identlist, the type they are declared to be has the same name, or the types they are declared to be has are declared equivalent (as in "**type** foo = bar"). (Standard Modula does not define compatibility of variables. The definition presented

here is *weak name equivalence.*) All integer variants (that is, shortint, midint, longint, integer) and names defined equivalent to them are compatible with each other. Likewise, all float variants (that is, ffloat, dfloat, float) and names defined equivalent to them are compatible with each other. Examples of variable declarations:

```
i,j,k : integer
n : dfloat
p,q : Boolean
ch : char
u : record
        s : bits
        a : vector
    end
s,t : bits
r : account
a : vector
m : matrix
w : array 1 : 10 of account
```

The syntactic construction of a designation of a variable is simply called "variable". It either refers to a variable as a whole, namely when it consists of the identifier of the variable, or to one of its components, when the identifier is followed by a selector. If a variable, say v, has a record structure with a field f, this component variable is designated by "v.f". If v has an array structure, its component with index i is designated by v[i].

```
variable = ident | variable "." ident | variable "[" indices "]".
indices = expression {"," expression}.
```

Examples of variables (see declarations above):

```
i      r.x   a[i]      m[i+1,j-1]      w[i].x      u.a[k]
```

8. Procedure declarations

Procedure declarations consist of a *procedure heading* and a block called the *procedure body*. The heading specifies the procedure identifier by which the procedure is called, and its formal parameters. The block contains declarations and statements.

There are two kinds of procedures, namely "proper procedures" and "function procedures". The latter are activated by a function call as a constituent of an expression and yield a result that acts as operand in the expression. The former are activated by a procedure call. The function procedure is distinguished in the declaration by the fact that the type of its result is indicated following the parameter list. Its body must contain an assignment to the procedure identifier that defines the value of the function procedure. The value of the function procedure must be of a simple type or an enumeration type; it may not be an array or record structure. However, it may not be dfloat.

There are three kinds of parameters, namely value, constant and variable parameters. The kind is indicated in the formal parameter list. *Value* parameters stand for a value obtained through evaluation of the corresponding actual parameter (expression) when the procedure is called. The type of the actual parameter must be compatible with the type of the formal parameter. (In particular, shortints and longints are compatible.) Only simple types may be passed by value. (Value parameters do not exist in Standard Modula.) *Constant* parameters are similar, except that assignments cannot be made to a constant formal parameter. *Variable* parameters correspond to actual variables, and assignment to them is permitted (see "Procedure calls"). Variable parameters must match exactly; shortint and longint do not match. Formal parameters are local to the procedure, that is, their scope is the program text that constitutes the procedure declaration.

All constants, variables, types, modules and procedures declared within the block that constitutes the procedure body are local to the procedure. The values of local variables, including those defined within a local module, are not defined upon entry to the procedure. Since procedures may be declared as local objects too, procedure declarations may be nested. Every object is said to be declared at a certain level of

nesting. If it is declared local to a procedure (or process) at level k, it has itself level k+1. Objects declared in the block that constitutes the main program are defined to be at level 0.

In addition to its formal parameters and local objects, objects declared in the environment of the procedure are also known and accessible in the procedure, unless the procedure declaration contains a use-list. In this case, only formal parameters, local objects and identifiers occurring in the use-list are known inside the procedure (see "Modules"). Standard objects are accessible in any case.

```
proceduredeclaration = procedure ident
        ["(" formalparameters ")"] [":" ident] ";" body.
formalparameters = section {";" section}.
section = [const | var] ident {"," ident} ":" formaltype.
formaltype = [array indextypes of] ident.
indextypes = identlist.
body = [uselist] block ident | forward.
uselist = use {identlist} ";".
block = {declarationpart} [statementpart] end.
declarationpart = const {constantdeclaration ";"} |
        type {typedeclaration ";"} |
        var {variabledeclaration ";"} |
        module ";" |
        proceduredeclaration ";" |
        processdeclaration ";" |
        value {ident "=" initialvalue ";"}.
initialvalue = (constantexpression | "{" {initialvalue} "}")
        [repetition].
repetition = {":" constantexpression}.
statementpart = begin statementsequence.
```

The identifier ending the procedure declaration must be the same as the one following the symbol **procedure**, that is, the procedure identifier. If the specifier **const** or **var** is missing in a section of formal parameters, then its elements are assumed to be value parameters.

An initialization part serves to assign values to variables declared in the same block. Braces indicate the structure of the assigned value, which must correspond to that of the initialized variable. Structured subparts of the value must be set off by

braces. Initialization parts can only occur in blocks at level 0, that is, in the main program and in modules declared in the main program. If a signal (see "Signals") is part of a structure that is being initialized, it must be initialized to 0. Uninitialized parts of an initialized structure are initialized to 0. (Standard Modula differs in syntax with respect to initializations.)

The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

If a formal type indicates an array structure, then only the types but not the bounds of the indices are specified. This formal type is known as a *generic array*. Legal bounds types include Boolean, integer (and its variants), char, and enumeration types.

If the body of a procedure is **forward**, then the procedure must be declared again later in the same compilation unit. Before it is declared again, however, it may be invoked. The purpose of **forward** declarations is to allow mutually recursive procedures. When the procedure declared with its body, the parameter list must be omitted. In addition, the return type for function procedures may be omitted. If it is not omitted, the type must agree with the type in the original declaration. (Standard Modula does not provide forward declarations.)

Procedures may be nested 8 deep. Up to 7 levels on nesting are allowed inside a process. (Standard Modula does not restrict the depth of procedure nesting.)

Examples of procedure declarations:

```
procedure readinteger(var x : integer);
var
        i : integer;
        ch : char;
begin
        i := 0;
        repeat
                readcharacter(ch)
        until ('0' <= ch) and (ch <= '9');
        repeat
                i := 10 * i + (integer(ch) - integer('0'));
                readcharacter(ch)
        until (ch<'0') or ('9' < ch);
end readinteger;

procedure writeinteger(x : integer);
var
        i,q : integer; (*assume x >= 0*)
        buf : array 1 : 10 of integer;
begin
        i := 0;
        q := x;
        writecharacter(' ');
        repeat
                i := i+1;
                buf[i] := q mod 10;
                q := q div 10
        until q = 0;
        repeat
                writecharacter(char(buf[i]+integer('0')));
                i := i-1
        until i = 0
end writeinteger;

procedure gcd(x,y : integer) : integer;
var
        a,b : integer; (*assume x,y > 0*)
begin
        a := x;
        b := y;
        while a <> b do
                if a < b then
                        b := b-a
                else
                        a := a-b
                end
        end;
        gcd := a
end gcd;
```

8.1. Standard procedures Standard procedures are predeclared and available throughout every program.

```
Proper procedures
        inc(x,n) = x := x + n  (n must be a constant expression)
        dec(x,n) = x := x - n  (n must be a constant expression)
        inc(x)   = x := x + 1
        dec(x)   = x := x - 1
        halt     = terminates the entire program
        printf(format,arguments)   (see below)


Function procedures
        Boolean := off (b1)
        Boolean := off (b1,b2)
                returns b1 (and b2) = []; b1, b2 of type bits
                not yet implemented
        Boolean := among (i,b)
                returns b[i]; b is of type bits
                not yet implemented
        integer := low (a)
        integer := low (a,n)
                returns low index bound of array a (dimension n)
        integer := high (a)
        integer := high (a,n)
                returns high index bound of array a (dimension n)
        integer := adr (v)
                returns address of variable v;
                implementation-dependent
        integer := size (v)
                returns size of variable v;
                implementation-dependent


procedures that interface with Unix:
        char := getchar
                returns a character from standard input
        integer := close(fd)
                closes a file
        integer := creat(filename,modes)
        integer := open(filename,modes)
        integer := read(fd,buffer,length)
        integer := seek(fd,location,mode)
        integer := write(fd,buffer,length)
        integer := syscall(number, arg1, arg2, ... )
```

Type transfer functions
integer(x) = ordinal of x in the set of values defined
by the type of x, converted to standard integer.
shortint(x) = ordinal of x in the set of values defined
by the type of x, converted to an 8-bit integer.
midint(x) = ordinal of x in the set of values defined
by the type of x, converted to a 16-bit integer.
longint(x) = ordinal of x in the set of values defined
by the type of x, converted to a 32-bit integer.
float(x) = floating point representation of the ordinal of
x in the set of values defined by the type of x.
ffloat(x) = floating point representation of the ordinal of
x in the set of values defined by the type of x.
dfloat(x) = floating point representation of the ordinal of
x in the set of values defined by the type of x.
char (x) = character with ordinal x.
bits (x) = bits with bit-pattern x.
<enum> (x) = enumeration element with ordinal x
(<enum> may be any enumeration type).

(Standard Modula does not provide a second argument to low and high, nor does it provide the "bits", "shortint", or "<enum>" type transfer functions, printf, nor any Unix interface procedures.)

Printf is a predeclared procedure useful for output. It is similar to the printf defined by the C language, and the following description is taken almost verbatim from *The C Programming Language* by B. W. Kernighan and D. W. Ritchie.

Printf takes an arbitrary number of arguments. The first argument must be a string (literal or array of char), specifying the desired output. Numeric, character, or string fields to be printed are marked in the format by field marks, which are of the form "%mn", where m is optional and may be:

- causes the converted argument to be left justified in its field

d...d a digit string specifying a minimum field width. The converted number will be printed in a field at least this wide, and wider if necesary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if a "-" preceded this digit string). If this digit string has a leading zero, the padding

character will be a zero, otherwise it will be a blank.

separates the width field from the precision field (below)

d...d a digit string (the precision) that specifies the maximum number of characters to be printed.

and n may be:

d	decimal output
o	octal output
x	hexadecimal output
s	string output
c	char output

The arguments that follow the format should correspond, in order, to the field specifications in the format. Example:

```
printf("The value of f(%d) is %d, called from %s0,
        i,f(i),caller);
```

Expressions of type "shortint", "midint", and "longint" should be converted to "integer" before being printed by printf.

The Unix interface procedures take arguments just as in C. These procedures may only be used when Modula is generating code for the VAX and the output of the compiler is linked with a Unix-compatible runtime. Syscall takes as a first argument the number of the system call. The "seek" call is the same as C's "lseek". Its second argument should be a longint.

9. Process declarations

A process declaration describes a sequential algorithm, including its local objects, that is intended to be executed concurrently with other processes. No assumption is made about the speed of execution of processes except that this speed is greater than zero.

A process declaration has the form of a procedure declaration, and the same rules about locality and accessibility of objects hold.

processdeclaration =
process ident ["(" formalparameters ")"] ";" body.

(Standard Modula allows a stack size to be associated with the process by a special syntax; our version uses a meaningful comment to adjust the compile-time parameter S. See "Vocabulary" and "Invoking mc".)

Parameters to processes may be of mode *value*, *variable*, or *constant*. Only *value* is safe, since *variable* parameters may be changed at any time by the caller or the new process, interfering with the other, and *constant* parameters may be changed by the caller, interfering with the process. In fact, if the caller is a process itself, it could terminate, leaving both constant and variable parameters pointing at deallocated space. (Standard Modula forbids a process from calling a process, however.)

The identifier at the end of the declaration must be the same as the one following the symbol **process**, namely the process identifier.

Processes must be declared at level 0; they cannot be nested or be local to procedures. Objects local to a process are said to be at level 1.

10. Expressions

Expressions are composed of operands (constants, variables and functions), operators and parentheses. They specify rules of computing values; evaluation of an expression yields a value of the type of the expression.

There are four classes of operators with different precedence (binding strength). Relational operators have the least precedence, then follow the adding operators, then multiplying operators, and finally the negation operator with highest precedence. Sequences of operators with equal precedence are executed from left to right.

Denotations of a variable in an expression refer to the current value of the variable. Function calls denote activation of a function procedure declaration (that is, execution of the statements that constitute its body). The result acts as an operand in the expression. The same rules about parameter evaluation and substitution hold as in the case of a procedure call (see "Procedure calls").

```
expression = simpleexpression [relation simpleexpression].
relation = "=" | "<>" | "<=" | "<" | ">" | ">=".
simpleexpression = ["+" | "-"] term {addoperator term}.
addoperator = "+" | "-" | or | xor.
term = factor {muloperator factor}.
muloperator = "*" | "/" | div | mod | and
factor = unsignedconstant | variable | functioncall |
  "(" expression ")" | not factor
functioncall = ident parameterlist.
```

Arithmetic operators (+ - * / **div mod**) apply to operands of type integer or float (or variants) and yield a result of the same type. Mixed float and integer arguments are allowed; the result is then float. Mixed precisions are allowed; the result is always the precision that the underlying machine provides for the operation. The operators + - * and / denote addition, subtraction, multiplication, and division. Division of integers leads to an integer result; the fraction is discarded. Division of floats leads to a float result. The monadic operators + and - denote identity and sign inversion. The operators **div** and **mod** yield a quotient q = x **div** y and r = x **mod** y such that x = q*y+r, 0≤r<y. The divisor (or modulus) y must be strictly positive. These two operators require integer (or variant) operands.

Example:
x = -15, y = 4
x/y = -3, x **div** y = -4, x **mod** y = 1.

Boolean operators (**or xor and not**) apply to Boolean operands and yield a result of type Boolean. The term a **and** b is evaluated as "**if** a **then** b **else** false", and the expression "a **or** b" is evaluated as "**if** a **then** true **else** b." Boolean operators can also be

applied to operands of type bits. The specified operation is then performed on all corresponding elements of the operands.

Relations yield a result of type Boolean. (<> <= >= stand for ≠ ≤ ≥ respectively). They apply to operands of the standard types integer, shortint, char, Boolean and bits (to the latter only = and <> apply), and of enumeration types. The two operands must be of compatible types.

Type transfer. A variable of one type may be cast to a different type by means of the **qua** operator. Thus one may write (see the declarations above) "w[2] **qua** vector [5]". **Qua** has the same precedence as "." and "[", and associates to the left. Conversions with **qua** are dangerous and should be used with care. Qua used to change from one variant of integer to another or from one kind of float to another will give garbage results. To convert an expression to a simple type (including an enumeration type), it is safer to use the name of the target type as a function call: "integer(ch)" or "color(3)". These conversions are checked for range. (Standard Modula only provides "integer" and "char", but not **qua**.)

Examples of factors:

27	i	(i+j+k)	**not** p	13 **qua** bs

Examples of terms:

i*k	i/(i-1)	i **div** 2	(i<j) **and** (j>k)	s **and** t

Examples of simple expressions:

i+j	i+5*k	-i	p **or** q	s **or** t

Examples of expressions:

(i+j)*(j+k)	i	k+5	i=j	t **xor** [0:7

(Given the variables declared in "Variable declarations", the first three examples in each line are of type integer, the fourth is of type Boolean and the fifth of type bits.)

11. Statements

Statements denote actions. Elementary statements are the assignment statement and the procedure call. Composite statements may be constructed out of elementary statements and other composite statements.

statement = assignment | procedurecall | processstatement |
ifstatement | casestatement | whilestatement |
repeatstatement | loopstatement | withstatement.

11.1. Assignments

An assignment denotes the action of evaluating an expression and of assigning the resulting value to a variable. The symbol := is called the *assignment operator* and is pronounced "becomes".

assignment = variable ":=" expression.

After an assignment is executed, the variable has the value obtained by evaluating the expression. The old value is lost. The variable must be type-compatible with the type of (the value of) the expression. If the type of the expression is some variant of integer, it will be coerced if necessary to match the type of the variable, which must be some integer or float type. Likewise, floating point expressions are coerced if necessary to another variant of floating point. Floating point cannot be coerced to integer. All coercions involve run-time range checks.

Examples of assignments:

```
i := 100
p := true
m[i,j] := 10*i+j
```

11.2. Procedure calls

A procedure call denotes the execution of the specified procedure, that is, of the statement part of its body. The procedure call must contain the same number of

parameters as the corresponding procedure declaration. An actual parameter corresponding (by its position in the parameter list) to a constant or value formal parameter may be an expression. The types of the actual and the formal parameters must be compatible. In the case of a constant parameter, the formal parameter is read-only; assignments to this parameter are prohibited. An actual parameter that corresponds to a variable parameter must be a variable. That variable is substituted for the formal parameter throughout the procedure body. Types must be identical, and if the actual parameter is an indexed variable, the index expressions are evaluated upon procedure call.

```
procedurecall = ident [parameterlist]
parameterlist = "(" parameter {"," parameter} ")".
parameter = expression | variable.
```

Examples of procedure calls:

```
inc(i,10)
sort(a,100)
```

11.3. Statement sequences

A sequence of statements separated by semicolons is called a statement sequence and specifies the sequential execution of the statements in the order of their occurrence.

```
statementsequence = statement {";" statement}.
```

11.4. If statements

If statements specify conditional execution of actions depending on the value of Boolean expressions.

```
ifstatement = if expression then statementsequence
        {elsif expression then statementsequence}
        [else statementsequence] end.
```

11.5. Case statements

Case statements specify the selective execution of a statement sequence depending on the value of an expression. First the case expression is evaluated, then the statement sequence with label equal to the resulting value is executed. The type of the case expression must not be structured.

It is an error if none of the labels has a value equal to the value of the case expression unless an otherwise clause is given. In that case, the statement list following **otherwise** is executed if no label is appropriate. (Standard Modula does not have an otherwise clause and differs in other minor ways from the syntax shown here.)

```
casestatement = case expression of case {case}
        [otherwise statementsequence] end.
case = caselabels ":" begin statementsequence end ";".
caselabels = constantexpression {"," constantexpression}.
```

11.6. While statements

While statements specify the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated before each execution of the statement sequence. The repetition stops as soon as this evaluation yields the value false.

```
whilestatement = while expression do statementsequence end
```

The most common error in Modula programming is due to the lack of a "for loop". You must use a while loop instead. The error is forgetting to increment the loop variable after each iteration.

11.7. Repeat statements

Repeat statements specify the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated after each execution of the statement sequence, and the repetition stops as soon as it yields

the value true. Hence, the statement sequence is executed at least once.

repeatstatement = **repeat** statementsequence **until** expression.

11.8. Loop statements

Loop statements specify the repeated execution of statement sequences. The repetition can be terminated depending on the values of possibly several Boolean expressions, called *exit conditions*.

loopstatement = **loop** statementsequence
{**when** expression [**do** statementsequence]
exit statementsequence} **end**.

Hence, the general form is

```
loop
        S1 when B1 do X1 exit;
        S2 when B2 do X2 exit;
        ...
        Sn when Bn do Xn exit;
        S
end
```

First, S1 is executed, then B1 is evaluated. If it yields the value true, X1 is executed and thereupon execution of the loop statement is terminated. Otherwise it continues with S2, etc. After S, execution continues unconditionally with S1.

All repetitions can be expressed by loop statements alone; the while and repeat statements express simple and frequently occurring cases.

11.9. With statements

The with statement specifies a list of record variables and a statement sequence to be executed. In these statements field identifiers of those record variables may occur without preceding qualification, and refer to the fields of the variable specified. Variables later in the list may refer to field identifiers of variables earlier in the list without qualification. (Standard Modula only allows a single variable, not a variable

list.)

withstatement = **with** variablelist **do** statementsequence **end**.

11.10. Process statements

A process statement expresses the initiation of a new process. Syntactically it corresponds to the procedure call. However, in the case of a procedure call, the calling program can be thought to be suspended until the procedure execution has been completed, whereas a program starting a new process is not suspended. Rather the execution of the started process may proceed concurrently with the continuation of the starting program.

processstatement = ident [parameterlist].

Whereas a process declaration defines a pattern of behavior, a process statement initiates the execution of actions according to this pattern. Reference to the same process declaration in several process statements initiates the concurrent execution of several processes according to the same pattern (usually according to different parameters).

Process statements may occur anywhere a statement is legal. (Standard Modula confines them to the body of the main program.)

12. Modules

A module constitutes a collection of declarations and a sequence of statements. They are enclosed in the brackets **module** and **end**. The module heading contains the module identifier, and possibly a *use-list* and a *define-list*. The former specifies all identifiers of objects that are used within the module and declared outside it. The latter specifies all identifiers of objects declare within the module that are to be used outside it. A module constitutes a wall around its local objects whose transparency is

strictly under control of the programmer. Objects local to a module are at the same level as the module.

module = moduleheading [definelist] [uselist] block ident.
moduleheading = [**interface**] **module** ident ";".
definelist = **define** identlist ";".

The identifier at the end of the module must be the same as the one following the symbol **module**, that is, the module identifier. For an explanation of the prefix **interface**, see "Interface modules". (Standard Modula allows "device" modules as well.) Identifiers that occur in the module's use-list are said to be *imported*, and those in the define-list are said to be *exported*.

If a type is defined local to a module and its identifier occurs in the define-list of the module, then only the type's identity, but none of its structural details becomes known outside the module. If it is a record type, the field names remain unknown; if it is an array type, index range and element types remain unknown outside. Hence, variables declared of a type that was exported in this way from a module can be used only by procedures declared within and exported from the same module. This restriction implies that if a module defines a type, it also must include the definition of all operators belonging to this type.

If a local variable occurs in the define-list of a module, it cannot be changed outside the module, that is, it appears as a read-only variable. (However, this restriction is relaxed across compilation units. See "Compilation Units".)

The statement sequence that constitutes the module body is executed when the procedure to which the module is local is called. If several modules are declared, then these bodies are executed in the sequence in which the modules occur. The bodies serve to initialize local variables. Example:

```
procedure P;
        module M1;
        define F1,n1;
        var n1 : integer;
        procedure F1(x : integer) : integer;
                begin
                        ...
                        inc(n1)
                        ...
                        F1 :=
                        ...
                end F1;
        begin
                n1 := 0
        end M1;

        module M2;
        define F2,n2;
        var n2 : integer;
        procedure F2(x  : integer) : integer;
                begin
                        ...
                        inc (n2)
                        ...
                        F2 :=
                        ...
                end F2;
        begin
                n2 := 0
        end M2;

begin (* use procedures F1 and F2; n1 and n2 count
        their calls and cannot be changed here *)
end P
```

In this example, the two statements "n1 := 0" and "n2 := 0" can be considered as prefixed to the body of procedure P. Within this body, assignments to these variables are prohibited.

Examples:

The following sample module scans a text and copies it onto an output character sequence. Input is obtained characterwise by a procedure "InChr" and delivered by a procedure "OutChr". Control characters are ignored with the exception of "\n" (new-line) and FileSeparator. They are both translated into a blank and cause the Boolean

variables "EndOfLine" and "EndOfFile" to be set respectively. Occurrences of FileSeparator are assumed to follow "\n" immediately.

```
module LineInput;

define
        Read, NewLine, NewFile, EndOfLine, EndOfFile, LineNumber;

use
        InChr, OutChr;

const
        FileSeparator = 34C;

var
        LineNumber : integer;
        ch : char; (* last character Read *)
        EndOfFile, EndOfLine : Boolean;


procedure NewFile;
begin
        if not EndOfFile then
                repeat
                        InChr(ch)
                until ch = FileSeparator;
        end;
        EndOfFile := false;
        LineNumber := 0
end NewFile;


procedure NewLine;
begin
        if not EndOfLine then
                repeat
                        InChr(ch)
                until ch = '0;
                OutChr('0);
        end;
        EndOfLine := false;
        inc(LineNumber)
end NewLine;
```

```
procedure Read(var x : char);
(* Assume not EndOfLine and not EndOfFile *)
begin
        while not EndOfLine do
                InChr(ch);
                OutChr(ch);
                case ch of
                '0:
                        begin
                                x := ' ';
                                EndOfLine := true;
                        end;
                FileSeparator:
                        begin
                                x := ' ';
                                EndOfLine := true;
                                EndOfFile := true;
                        end;
                otherwise
                        x := ch
                end (* case *)
        end (* while *)
end Read;

begin (* LineInput *)
        EndOfFile := true;
        EndOfLine := true
end LineInput
```

The next example is a module that operates a disk-track reservation table and protects it from unauthorized access. A function procedure "NewTrack" yields the number of a free track that is being reserved. Tracks can be released by calling procedure "ReturnTrack".

```
module TrackReservation;

define NewTrack, ReturnTrack;

const
        TableLength = 64;
        WordLength = 16;
        (* There are TableLength * WordLength tracks. *)

var
        Reserved: array 0 : TableLength-1 of bits;
```

```
procedure NewTrack : integer;
(* Reserves a new track, yields its index as function
result, if a free track is Found, and -1 otherwise *)
var
        TableIndex,BitIndex : integer;
        Found : Boolean;
begin
        Found := false;
        TableIndex := TableLength;
        repeat
                dec(TableIndex);
                BitIndex := WordLength;
                repeat
                        dec(BitIndex);
                        Found := not Reserved[TableIndex,BitIndex];
                until Found or (BitIndex=0);
        until Found or (TableIndex=0);
        if Found then
                NewTrack := TableIndex*WordLength+BitIndex;
                Reserved [TableIndex,BitIndex] := true
        else
                NewTrack := -1
        end
end NewTrack;


procedure ReturnTrack(TrackNumber : integer);
(* Assume 0 <= TrackNumber < TableLength * WordLength. *)
begin
        Reserved[TrackNumber div WordLength,
                TrackNumber mod WordLength] := false
end ReturnTrack;


procedure TrackInit;
var
        TrackNumber : integer;
begin
        TrackNumber := TableLength; (*mark all tracks free*)
        repeat
                dec(TrackNumber);
                Reserved[TrackNumber] := []
        until TrackNumber = 0
end TrackInit;


begin (* TrackReservation *)
        TrackInit;
end TrackReservation
```

13. Interface modules

The *interface module* is the facility that provides exclusion of simultaneous access from several processes to common objects. Variables that are to establish communication or data transfers between processes are declared local to an interface module. They are accessed via *interface procedures* also declared locally. and exported from the interface module. If any process has called any such procedure, another process calling the same or another one of these procedures in the same interface module is delayed until the first process has completed its procedure or starts waiting for a signal (see "Signals").

An interface module is syntactically distinguished from regular modules by the prefix symbol **interface**.

Interface procedures may call procedures declared outside the interface module. (Standard Modula only allows such calls on standard procedures.) Exclusion on the interface module is maintained during the invocation of such procedures. If the called procedure is in a different interface module, then that interface module also becomes locked. It is possible for the procedure to invoke another procedure, either directly or indirectly, within an interface module already locked by the same process; such invocation is legal. If a process in an interface procedure invokes another interface procedure that then executes a "send" or "wait" (see "Signals"), only the last-acquired lock is released. Deadlock can therefore easily result from invocation of interface procedures from interface procedures.

Examples of interface modules are given below.

13.1. Signals

In general, processes communicate via common variables, usually declared within interface modules. However, it is not recommended that synchronization be achieved

by means of such common, shared variables. A delay of a process could in this way be realized only by a "busy waiting" statement, that is, by polling. Instead, a facility called a *signal* should be used.

Signals are introduced in a program (usually within interface modules) like other objects. In particular, the syntactic form of their declaration is like that of variables, although the signal is not a variable in the sense of having a value and being assignable. (A signal is more like a constant than a variable.) Signals must be declared at static level 0, but need not be inside interface modules. Signals may only be passed as parameters in *variable* mode. There are only two operations and a test that can be applied to signals, represented by three standard procedures.

1. The procedure call "wait(s,r)" delays the process until it receives the signal s. The process is given delay rank r, where r must be a positive-valued integer expression. The abbreviation "wait(s)" may be used for "wait(s,1)".
2. The procedure call "send(s)" sends the signal s to that process that has been waiting for s with least delay rank. If several processes are waiting for s with same delay rank, the process that has been waiting longest receives s. If no process is waiting for s, the statement "send(s)" has no effect.
3. The Boolean function procedure "awaited(s)" yields the value "true" if there is at least one process waiting for signal s, "false" otherwise.

If a process executes a call on wait within an interface procedure, then other processes are allowed to execute other such procedures, although the waiting process has not completed its interface procedure. If a send statement is executed within an interface procedure, and if the signal is sent to a process waiting within the same interface module, then the receiving process obtains control over the module and the sending process is delayed until the other process has completed its interface procedure. (A send statement outside the interface procedure does not pre-empt a pro-

cess executing within the interface procedure, on the other hand.) Hence, both the wait and send operations must be considered as singular points or enclaves in the interface module, and are exempted from the mutual exclusion rule.

Send operations may be applied to imported signals. (Standard Modula forbids such uses of "send".)

There is a predeclared signal "panicsig". If all processes are blocked waiting for signals, the runtime will execute "send(panicsig)". If all processes are still blocked, debugging output is displayed and the program ended abnormally. (Standard Modula does not include "panicsig".)

Signals may be declared and used outside interface modules, but good programming practice limits the number of such unbound signals.

Examples of interface modules with signal operations:

```
interface module ResourceReservation;

define Semaphore, P, V, Init;

type
        Semaphore =
                record
                        Taken : Boolean;
                        Free : signal
                end;

procedure P (var s : Semaphore);
begin
        if s.Taken then
                wait (s.Free)
        end;
        s.Taken := true
end P;

procedure V (var s : Semaphore);
begin
        s.Taken := false;
        send (s.Free)
end V;

procedure Init (var s : Semaphore);
begin
        s.Taken := false
end Init;

end ResourceReservation;
```

```
interface module BufferHandling;

define Get, Put, Empty;

const
        BufferMax = 256;

var
        BufferCount, InIndex, OutIndex : integer;
        NonEmpty, NonFull : signal;
        Buffer : array 1 : BufferMax of char;

procedure Empty : Boolean;
begin
        Empty := BufferCount = 0
end Empty;

procedure Put (ch : char);
begin
        if BufferCount = BufferMax then
                wait (NonFull)
        end;
        inc(BufferCount);
        Buffer[InIndex] := ch;
        InIndex := (InIndex mod BufferMax) + 1;
        send(NonEmpty)
end Put;

procedure Get(var ch : char);
begin
        if BufferCount = 0 then
                wait(NonEmpty)
        end;
        dec(BufferCount);
        ch := Buffer[OutIndex];
        OutIndex := (OutIndex mod BufferMax) + 1;
        send(NonFull)
end Get;

begin (* BufferHandling *)
        BufferCount := 0;
        InIndex := 1;
        OutIndex := 1
end BufferHandling;
```

```
interface module DiskHeadScheduler;

define Request, Release;

use CylinderMax; (* Number of cylinders *)

var
        HeadPosition : integer;
        Up, Busy : Boolean;
        UpSweep, DownSweep : signal;

procedure Request(Destination : integer);
begin
        if Busy then
                if HeadPosition < Destination then
                        wait(UpSweep, Destination)
                else
                        wait(DownSweep, CylinderMax-Destination)
                end
        end;
        Busy := true;
        HeadPosition := Destination
end Request;

procedure Release;
begin
        Busy := false;
        if Up then
                if awaited(UpSweep) then
                        send(UpSweep)
                else
                        Up := false;
                        send(DownSweep)
                end
        else
                if awaited(DownSweep) then
                        send(DownSweep)
                else
                        Up := true;
                        send(UpSweep)
                end
        end
end Release;

begin
        HeadPosition := 0;
        Up := true;
        Busy := false
end DiskHeadScheduler
```

14. Compilation units

Large programs may be divided into individual compilation units and compiled separately. (Standard Modula does not support separate compilation.) Each compilation unit must consist of exactly one module, although that module may contain other modules within it.

compilation unit = module "."

14.1. Import and export

All variables, types, procedures, and processes that are imported into a compilation unit must also be declared in that compilation unit. No checking is performed to insure that the declarations are accurate (but see "Source File Switching", below).

Types imported from another compilation unit do not suffer the visibility restrictions that apply to types imported into modules; all subparts of structured types are visible. It is therefore unnecessary to list imported types in the use-list; just declaring them is adequate.

Variables imported from another compilation unit do not suffer the read-only restriction that applies to variables imported into modules.

Procedures and processes imported from another compilation unit should be declared with the same header as in their home compilation unit. The body of the procedure or process should be replaced by the reserved word **external**.

A process or procedure exported from a compilation unit may also have an **external** declaration in that compilation unit. In this case, the word **external** is treated identically to **forward**. (The purpose of this feature is explained below, under "Source File Switching".)

14.2. Source File Switching

Our implementation of Modula uses the C pre-processor, which makes it is very convenient to create files of external declarations that are shared among all the modules that might use them.

To include "otherfile" within a compilation unit, insert the following line:

```
#include "otherfile" (* the double-quotes are required *)
```

This line must start at the left margin.

Modula allows **const**, **type**, **var**, **value**, **procedure** and **process** declarations to appear in any order, so long as every identifier is declared before it is first used. In addition, the keywords **const**, **var**, etc. may appear several times in the same module. Therefore, included files may intersperse various sorts of definitions. On the other hand, the **define** and **use** clauses must appear only once and in that order.

The following examples use the convention that if a module is called "foo", "foo.m" contains its code, "foo.h" contains all declarations that it exports (types, variables, procedures and processes, with the latter two containing the body "**external**"), and "foo.d" contains a list of exported identifiers, separated by commas. Then the file "foo.m" will start like this:

```
module foo;
define
#include "foo.d"
;
#include "foo.h"
< other included declarations used by foo >
```

and module "bar" that uses those declarations will start like this:

```
module bar;
define
#include "bar.d"
use
#include "foo.d"
;
#include "bar.h"
#include "foo.h"
```

15. Programs

A Modula program is a set of compilation units. Exactly one compilation must contain a module named "main", which is the module that begins executing (as a process) when the program starts. All module bodies in "main" are executed (in the order they appear), but the bodies of the other modules are not executed. In order to invoke the body of a module in another compilation unit, the name of that module must be imported and declared to be a procedure with no parameters. Invoking that procedure causes the body of that module to be executed.

If you want a process to wait before proceeding until all initialization is finished, the process should wait on a go-ahead signal as its first action.

A program terminates when either all processes have terminated or when the program executes a halt statement.

16. Macros and conditional compilation The compiler uses the C compiler's preprocessor, which supports macros and conditional compilation. The conditional compilation facility is useful for selectively including debugging code. Various levels of debugging can also be supported. For example:

```
#if DEBUG
        (* here are some debugging statements *)
#if DEBUG > 2
        (* here are some more verbose debugging statements *)
#if DEBUG == 5
        (* here are some very specific debugging statements *)
#endif
#endif
#endif
```

When you are debugging, the file might start with the line

```
#define DEBUG 3
```

Alternately, one may pass a debug flag to the pre-processor:

```
vmc -DDEBUG=3
```

(See "Invoking Modula".)

17. The compiler and runtime library

The compiler and runtime are in a state of flux as bugs are reported and fixed. The information in this section is therefore likely to change. In what follows, the local computers are referred to as "crystal", "uwvax", "slovax", and "dbvax".

17.1. Files

The most recent (that is, experimental) version of the compiler may be found on crystal in the directory

```
/usr/crystal/mc/test
```

The compilers for the VAX and PDP-11 are called "vaxmod" and "pdpmod", respectively. Previous released versions of the compiler may be found on crystal in

```
/usr2/staff/nhall/keith/pr/versx.y
```

where x is the major release number, and y is the minor release number.

The currently released versions of the compilers and libraries are in the files

```
compilers:
        dbvax: directory /usr/crystal/mc/bin
        crystal: directory /usr/crystal/mc/bin
        slovax: directory /usr/lib/mc/bin
                vaxmod              (VAX)
                pdpmod              (PDP-11)
                vaxmod.old   (previous; VAX)
                pdpmod.old   (previous; PDP-11)

runtime libraries:
        dbvax: directory /usr/crystal/mc/lib
        crystal: directory /usr/crystal/mc/lib
                also, directory /compat/v7/usr/crystal/mc/lib
        slovax:  directory /usr/lib/mc/lib
                pdpblib.a           (bare PDP-11)
                pdpulib.a      (V7 Unix PDP-11)
                pdpclib.a      (Charlotte PDP-11)
                vaxulib.a      (VAX)

startup files, to be loaded first:
        dbvax: directory /usr/crystal/mc/lib
        crystal: directory /usr/crystal/mc/lib
                also, directory /compat/v7/usr/crystal/mc/lib
        slovax:  directory /usr/lib/mc/lib
                vaxustart.o   (VAX)
                pdpustart.o   (V7 Unix PDP-11)
                pdpbstart.o   (bare PDP-11)
                pdpcstart.o   (Charlotte PDP-11)
```

The Charlotte version of the runtime does not allow processes or all consequences of processes, including interface modules and signals.

Experimental runtime libraries and startup routines are on crystal in the files

```
/usr/crystal/mc/test
        libmseqP.a    (PDP library without processes)
        libmiP.a              (bare PDP library with processes)
        libmP.a               (PDP under Unix library with processes)
                (similar libraries for the VAX are forthcoming)
        startPB.o             (startup for bare PDP)
        startPBI.o    (startup for bare PDP with interrupts)
        startPU.o             (startup for PDP under Unix)
                (similar startups for the VAX are forthcoming)
        putcharPB.o (putchar routine for bare PDP)
        putcharPU.o (putchar routine for the PDP under Unix)
```

The naming convention for the experimental versions of the libraries will replace that for the released versions when the experimental versions are complete and debugged.

On machines other than crystal, replacement will wait until the end of the Spring 1983 semester.

17.2. Invoking Modula To use either the VAX or PDP-11 versions of the Modula compiler, one may invoke the compiler directly. In order to compile a single compilation unit, the invocation is:

```
/lib/cpp <source.m> <source.cpp>
<compiler> <source.cpp> <source.s> <compile options>
<assembler> <flags> -o <source.o> <source.s>
<load> <startup file> <source.o files> <library> -lc
(runnable program is in a.out)
```

The first line calls the standard C macro pass. The second line invokes the compiler. The third line assembles the output of the compiler. If the VAX is the target machine, <assembler> is "as", and <flags> is "-". If the PDP-11 is the target machine, <assembler> is "v7run -/compat/v7 /bin/as", and <flags> is "-u". The fourth line is the loading step; it should be invoked after all the source files have been compiled. If the VAX is the target machine, <load> is "ld". If the PDP-11 is the target machine, <load> is "v7run -/compat/v7 /bin/ld". In this case, the startup file and the library file must be expressed as an absolute pathname starting from the directory /compat/v7 or a relative pathname starting from the current working directory.

The available <compile options> are:

```
s: produce source listing
w: produce banner identifying the compiler
a: comment assembler code
```

In addition, the following options are set by default:

```
R: generate runtime checks (on)
B: generate calls to process switching in each loop (on)
S: stack size for each process (in bytes for PDP, default=510;
                in longwords for VAX under Unix, minimum=default=1000)
O: improve code by recognizing common subexpressions (off)
u: improve code by using better register allocation scheme (off)
```

Any of these eight options may be changed at compile time by meaningful comments.

(See "Vocabulary".) (Turning the "w" option on or off at compile-time has no effect.)

A much simpler invocation, without quite the same power, is available:

vmc [flags] [files] (for VAX version)
pmc [flags] [files] (for PDP-11 version)

All the ".m" files in the list will be compiled to the equivalent ".o" files, and all the ".o" files will be linked together into "a.out".

Available flags:

h: print a help message
c: don't remove .o file(s)
o: next argument is to be used for the output of ld
S: do not remove intermediate .s files
t: just compile - don't assemble or link
v: be verbose
L: produce source listing
A: comment assembler code
j: don't strip load module
T: use test version of the compiler (only for crystal)
R: next argument is an alternative runtime support package
D: remainder of string (until blank) defined to preprocessor
U: remainder of string (until blank) undefined to preprocessor
I: remainder of string is a directory searched for "includes"
O: improve code by recognizing common subexpressions
u: improve code by using better register allocation scheme