AN INTRODUCTION TO
MODULAR PROGRAMMING FOR PASCAL USERS

by

Robert P. Cook and Stephen J. Scalpone

Computer Sciences Technical Report #372

November 1979

AN INTRODUCTION TO MODULAR PROGRAMMING FOR PASCAL USERS

1. BLOCKS

In this discussion, capital letters will be used for syntactic constructs and lower case for language keywords. In text, both notations will be distinguished by double quotes. A "BLOCK", then, denotes a lexical scope for a series of "DECLARATION"s followed by a "STATEMENT_SEQUENCE". The repetition of a construct is represented as '[ ]...'; for example, "[DIGIT]..." indicates a sequence of one or more "DIGIT"s. Finally, '{ }...' is used to indicate zero or more instances of a string.

```
PROGRAM = module IDENTIFIER;
                [DEFINE_LIST; ]
                BLOCK IDENTIFIER.
```

A "PROGRAM" is the unit of compilation and must follow the previous definition. If the "IDENTIFIER" is "main", execution will be started with this module. The two "IDENTIFIER"s must match and the "." indicates the end of the program text. In general, all lexical scopes are named and must terminate with an identical name; thus, the text of a block can be easily delineated by the block identifier.

This language definition should be considered as experimental and is subject to change from time to time. For an extensive set of examples and some historical perspective, the description of Modula by Wirth[1 2 3] should be consulted.

```
BLOCK    = [USE_LIST; ]
           {BLOCKHD }...
           [begin STATEMENT_SEQUENCE ]
           end

BLOCKHD = const {IDENTIFIER = CONSTANT_EXPRESSION;}... |
          type {IDENTIFIER = [^] TYPE;}...              |
          var {IDENTIFIER_LIST: [^] TYPE;}...           |
          value {IDENTIFIER = INITIAL_VALUE;}...        |
          MODULE_DECLARATION;                           |
          PROCEDURE_DECLARATION;                        |
          PROCESS_DECLARATION;
```

Unlike PASCAL, the declarations can occur in any order but not in the body of the "BLOCK". Again, note that the "IDENTIFIER" following the "end" of the "BLOCK" must match the name of the "module", "process", or "procedure" in which it occurs. The comment convention "(*...*)" is the same as Pascal, except that any sequence of statements may be commented out by enclosure with a comment.

```
IDENTIFIER_LIST  = IDENTIFIER {,IDENTIFIER}...
IDENTIFIER       = LETTER {LETTER|DIGIT}...
LETTER           = a|b|...|z|A|B|...|Z|_

CONSTANT         = CONST_IDENTIFIER | INTEGER | CHAR | STRING |
                   BIT_CONSTANT | true | false | nil
INTEGER          = [DIGIT]... | [OCTAL_DIGIT]...b
CHAR             = 'CHARACTER' | [OCTAL_DIGIT]...c
STRING           = '[CHARACTER]...'
```

Thus, integers can be expressed in octal as well as decimal. In addition, a single character can also be represented as its octal equivalent. A " ' " can be represented in a string constant as a " '' ". A "STRING" of length three is considered the same "TYPE" as an array with three "CHAR" elements. A named constant can be declared by using the "const" declaration in which case each use of the "const" IDENTIFIER will be replaced by its associated constant value.

```
TYPE = integer | boolean | char | bits | signal |
       ENUMERATION | ARRAY | RECORD
```

Note that subrange types are not currently implemented. All variables must be declared before use, which may require using a "forward" declaration for "procedures" and "processes".

"integer" is a signed integer which occupies one machine word, as do "bits" and "signal" variables. "boolean" variables can take on a value of "true" or "false" and occupy one machine word. "char" variables are stored in 8 bits and have single characters as values. Currently, all variables are packed in declaration order, so care must be taken with "char" variables to insure that succeeding declarations start on word boundaries(two characters per word). Pointer variable(^TYPE) are defined as in Pascal except that the "new" procedure is not implemented. A "bits" variable is a wordlength-size boolean value which can be manipulated by any boolean operator; thus, "a:= b or c;" would logically "or" the wordlength values in "b" and "c", storing the result in "a". "bits" can be set to "true" or "false" in any selected positions by using the following notation.

```
BIT_CONSTANT = "[" BIT_LIST "]"
BIT_LIST     = CONSTANT_EXPRESSION {,CONSTANT_EXPRESSION}...
```

As an example, "a:=[Ø,4];" would set bits Ø and 4 to "true" and all other bits to "false". Bit zero is always the rightmost bit. In addition, "true" can be used for an all ones value and "false" can be used for zero.

```
ENUMERATION = (IDENTIFIER_LIST)

ARRAY = array INDEX_LIST of [^] TYPE
INDEX_LIST = INDEX {,INDEX}...
INDEX = CONSTANT [:CONSTANT]
```

Enumerated types are identical to Pascal; arrays are not the same, however. Note the absence of "[ ]". If the lower "INDEX" is omitted, a default value is chosen which represents the constant with ordinality zero that matches the upper "INDEX". The component type of the array can be anything, including other arrays. In addition, strings may not be used as array bounds. Finally, an array of arrays is distinct from a single multidimensional array declaration.

```
RECORD = record FIELD_LIST [;FIELD_LIST]... end
FIELD_LIST = IDENTIFIER_LIST : [^] TYPE
```

Records are identical to PASCAL, except that variant parts are not implemented.

TWO VARIABLES ARE OF THE SAME "TYPE" IF THEY WERE DECLARED WITH THE SAME TYPE NAME.

In general, operators require their operands to match in "TYPE".

```
value {IDENTIFIER = INITIAL_VALUE;}...
INITIAL_VALUE      = CONSTANT  |
                     (INITIAL_VALUE {,INITIAL_VALUE}... )  |
                     "[" REPEAT_COUNT "]" INITIAL_VALUE
REPEAT_COUNT       = CONSTANT
```

An initialization part can be used to assign initial values to a module's variables. It cannot occur inside a procedure or process declaration. The "INITIAL_VALUE" specifies a storage template which must not exceed the variable's storage area in size. For convenience, an integer "REPEAT_COUNT" may be used to replicate any value or list of values. Pointers, signals and variables declared with a protected type cannot be initialized with this construct.

2. STATEMENT SEQUENCE

The language is structured so that once a program is constructed, additional statements can be added at any point with no other changes necessary, such as adding "begin" - "end" pairs. To maintain this convention, it is advisable, although not strictly necessary, to terminate each "STATEMENT" with a ";".

```
STATEMENT_SEQUENCE = STATEMENT {;STATEMENT}...

STATEMENT = ASSIGNMENT | PROCEDURE | PROCESS |
            IF | CASE | WHILE | REPEAT | LOOP | WITH

ASSIGNMENT = VARIABLE := EXPRESSION

PROCEDURE = IDENTIFIER [PARAMETERS]
PARAMETERS = (EXPRESSION {,EXPRESSION}...)
```

The procedure call must contain the same number of parameters and with the same types as in the procedure declaration. The arguments are evaluated from left to right before the procedure is called. If the procedure returns a value, it is ignored. Arguments may be passed by value ("const") or by address ("var"). Only integers, booleans, enumerations, characters, or bits can be used as value arguments and only values of these types can be returned from functions. All other constructs must be declared as address parameters. The builtin procedure statements are as follows:

```
halt                              =       terminates the program
inc(x)                            =       x:=x+1
dec(x)                            =       x:=x-1
inc(x,EXPRESSION)                 =       x:=x+EXPRESSION
dec(x, EXPRESSION)                =       x:=x-EXPRESSION
sys(any number of arguments)      outside interface
```

Since no I/O facilities are defined, "sys" allows the user to build anything desired. It can also be used as an "integer" function. "sys" generates a library call with the first argument passed by value and the remaining arguments by address. It is intended that "sys" be used to implement and isolate machine dependent functions; it should be used with care.

PROCESS = IDENTIFIER [PARAMETERS]

A process statement starts a new process running in parallel with the current process. Initially, the only process executing is the main program. Also, calls on processes with "var" argu-

ments are restricted to the main program to prevent dangling reference problems.

```
IF = if EXPRESSION then STATEMENT_SEQUENCE
     {elsif EXPRESSION then STATEMENT_SEQUENCE}...
     [else STATEMENT_SEQUENCE]
     end
```

All "IF" statements must terminate with an "end". The "else" matches the nearest previous "then". "elsif" is a short-hand for "else if" and has the additional advantage that only one terminating "end" is required. The "then" part is selected when "EXPRESSION" is not false.

```
CASE = case EXPRESSION of CASES {;CASES}... end
CASES = CASE_LABELS: begin STATEMENT_SEQUENCE end
CASE_LABELS = CONSTANT_EXPRESSION {,CONSTANT_EXPRESSION}...
```

All "CASE" statements must terminate with an "end". The "CASELABELS" must evaluate to a constant which matches the type of the case selector expression. If the expression does not match any of the labels, an error message will be printed at run-time.

```
WHILE = while EXPRESSION  do STATEMENT_SEQUENCE end
```

The statements are executed as long as "EXPRESSION" is not false at the beginning of each iteration.

```
REPEAT = repeat STATEMENT_SEQUENCE until EXPRESSION
```

The statements are executed as long as "EXPRESSION" is false

at the end of each iteration.

```
LOOP = loop STATEMENT_SEQUENCE
            {when EXPRESSION  [do STATEMENT_SEQUENCE ] exit
             STATEMENT_SEQUENCE}...
            end
```

All "LOOP" statements must terminate with an "end". The "when" clauses may be interspersed at will within the loop body; thus, the loop body is executed sequentially and repetitively until one of the "when" clauses becomes true. At this point, the optional "do" statements will be executed and the loop terminated.

```
WITH     = with SELECTOR do STATEMENT_SEQUENCE end
SELECTOR = RECORD_VARIABLE | SIMPLE_VARIABLE
```

Within the "with" statement, the fields of the specified record variable can be denoted by their field identifier only, i.e. without preceding them with the denotation of the entire record variable. The "with" statement effectively opens the scope containing the field identifiers of the specified record variable, so that the field identifiers may occur as variable identifiers.

If a simple variable is listed, it is interpreted by the compiler as an indication that the variable occurs frequently in the body of the "with" statement. The compiler will then try to keep the variable in a fast storage medium. If this necessitates making a copy of the variable's value, the variable will be updated at the end of the "with" statement.

3. EXPRESSIONs

The components of an "EXPRESSION" can be record references (a.b.c), array references (a[x,y]), constants, simple variables, function references (sin(x)), parenthesized expressions (a*(b+c)), and operators. The operators are listed below along with the operand types and precedence levels.

not	boolean, bits
unary +	integer
unary -	integer
*	integer
div	integer
mod	integer
and	boolean, bits
+	integer
-	integer
or	boolean, bits
xor	boolean, bits
=	array, record, string, char, integer, enumeration, boolean, bits
<>	" "
<=	enumeration, char, integer
>=	" "
<	" "
>	" "
:=	array, record, string, char, enumeration, integer, boolean, bits

The only builtin functions are "ord", "char" and "addr".

4. PROCEDURE DECLARATION

Procedures are the same as Pascal, except for functions, and have the following format:

```
PROCEDURE_DECLARATION = procedure IDENTIFIER [(FORMALS)]
                                        [: [^] IDENTIFIER]; [forward;]
                        BLOCK IDENTIFIER
```

The "BLOCK" "IDENTIFIER" must match the procedure name. The "BLOCK", representing the body of the procedure, cannot contain declarations for processes or interface modules. The ": IDENTIFIER" indicates a function and specifies a type name for the returned value. The returned value for functions must be set by assignment to the procedure's "IDENTIFIER". Procedures can only exit at the end of the "BLOCK" and can be recursive. The "USE_LIST" is defined under "MODULE_DECLARATION". Also, only one level of nesting is allowed for procedure declarations.

```
FORMALS = SECTION {;SECTION}...
SECTION = [const | var] IDENTIFIER_LIST : [^] TYPE
```

"const" parameters can have an enumeration, "integer", "char", "boolean" or "bits" type. The corresponding argument must match this type and is completely evaluated before the procedure call; in other words, the parameter cell contains the value of the argument. For "var" parameters the cell contains the address of the argument and any reference to the parameter name is a reference to the argument variable. If "forward" is specified, the

remainder of the declaration is omitted and must occur later in the block as follows:

```
procedure IDENTIFIER;
          [USE_LIST;]
          BLOCK IDENTIFIER
```

5. PROCESS DECLARATION

A "process" is an entity, not found in Pascal, which can be used to explore areas of parallel programming.

```
PROCESS_DECLARATION = process IDENTIFIER [(FORMALS)];  [forward;]
                      BLOCK IDENTIFIER
```

A process is an independent execution entity which is initiated by a "PROCESS" statement. The number of concurrent processes at execution time is limited only by the memory of the host computer. A process declaration has the form of a procedure declaration, and the same rules about locality and accessibility of objects hold. A process is terminated by executing the "end" statement in the "BLOCK".

6. MODULE DECLARATION

A module defines a lexical scope at compile time and is used to encapsulate data structures, and the procedures and processes to manipulate them.

```
MODULE_DECLARATION = [interface] module IDENTIFIER;
                     [DEFINE_LIST;]
                     [USE_LIST;]
                     BLOCK IDENTIFIER

DEFINE_LIST = define IMEX_LIST
USE_LIST    = use    IMEX_LIST
IMEX_LIST   = IDENTIFIER [*] {,IDENTIFIER [*] }...
```

The "BLOCK" that constitutes the module body is executed when the procedure or process to which the module is local is activated. If several modules are declared in parallel, the execution sequence is in declaration order. Thus, the module body is used to start the main program and to initialize local variables. Note that module initialization will occur only for the "main" program; any initialization in separate compilations must be done explicitly. Every separately compiled module has an implicit procedure associated with it. This procedure has the same name as the module name and can be called to execute the initialization code for the module. Note that the module name must occur in a "use" list and a procedure heading before it can be used for initialization.

The "define" and "use" lists specify which variables to export or import, respectively. Normally, all global variables are accessible within a module. However, the appearance of a "define" or "use" list closes the scope of reference to only those variables in the "use" list and locals. Each identifier in a "use" list must be declared in a global block and will have its declaration copied into the block of the "use". The "define" statement copies definitions of local variables into the scope of

the enclosing block. Thus, "define" and "use" force the programmer to specify the external interface for each block in which they are used.

An "*" following any name indicates a reference to a symbol which will be used external to the current compilation. "define" exports a definition for the referenced symbol to the load-time environment; while "use" imports a symbol from another compilation. Only variables, processes and procedures may be used as external symbols. In addition, symbols in the "use" list must also be followed by a declaration. For procedures and processes, only the heading need be declared. Once an imported variable is declared, it can be referenced in succeeding modules by inclusion in a "use" list; no further declaration is necessary.

Each variable exported within a compilation is made read-only to the scopes outside its block of definition. Also, exported types are treated as opaque; that is, variables declared using the type name are read-only and only the major name can be referenced. For example, an exported "record" type could only be referenced as a record variable; the components would be invisible. Finally, any exported variable or instance of an exported type can be passed as a "var" argument to an exported procedure from the same block to manipulate its value. Variables exported from, or imported to, the program are an exception and are read/write. Protection for external variables is obtained by exporting only procedure names while encapsulating the data.

ANY LOGICAL UNIT OF A PROGRAM, TOGETHER WITH ITS DATA, SHOULD ALWAYS BE MODULATED. This serves as a useful documentation convention and also protects variables from unwanted side-effects.

7. Interface Module

Only procedures or variables can be declared within the scope of an interface module since the intent is for no process to execute within such a module for any length of time. Interface modules have the property that only one process can be executing within the same module at the same time; thus, simultaneous accesses to encapsulated data and procedures are prohibited. If a process attempts to enter a "busy" interface module, it is delayed until no process is executing within the module. Note that this implies that waiting processes in an interface module do not exclude newcomers. In addition, the order in which waiting processes are selected for entry is non-deterministic.

8. Signals[1]

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, i.e. 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. There are only two operations and a test that can be applied to signals. They are 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. "wait(s)" is a short form for wait(s,1).

2. The procedure call "send(s)" sends the signal s to that process which had been waiting for s with largest delay rank. If several processes are waiting for s with the same delay rank, that process receives s which had been waiting longest. 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 wait staement within an interface procedure, then other processes are allowed to execute other such procedures, even though the waiting process had 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. Hence, both the wait and send operations must be considered as 'singular points' or enclaves in the interface module, which are exempted from the mutual exclusion rule.

If a signal variable is exported from a module, then no send

operations can be applied to it outside the module."

9. The PDP11 Implementation

The compiler is a 3000 line C program which runs as a shared processor under the UNIX operating system. The size is 60K bytes of which 33K bytes are shared by all users of the system. The compilation rate is approximately 3000LPM. The conventions for invoking the compiler under UNIX are illustrated in a later section. The generated code is operating system independent and can be loaded with a UNIX kernel for class use or program development. A stand-alone kernel is also available for bare machine programming. The stand-alone kernel can be used on any PDP11 and also on LSI11s.

Since the "sys" call is machine independent; a special run-time package has been constructed to interface with the operating system. This package may be deleted or augmented at the pleasure of the user community. The system I/O calls are implemented as follows:

sys(1, list of integers)	prints integers in octal followed by a carriage return.
sys(2, C format, list of integer, char, boolean, bits)	prints list with C format followed by a carriage return.
sys(3, C format, list of integer, char, boolean, bits)	prints list with C format
sys(4, C format, string)	prints string with C format followed by a carriage return.
sys(5, C format, string)	prints string with C format

The "C format" must be a string constant with the following

forms:

%wd	decimal in field of width w
%wo	octal in field of width w
%c	single character
%s	string of characters (can only be used with sys(4) or sys(5) and then only once)

In addition the following operating system interface is provided:

sys(6, opcode, argument list)

opcode	meaning	argument types	returned value
1	change directory	string	Ø-no error
2	change mode	string, integer	Ø-no error
3	close file	integer	Ø-no error
4	create a new file	string, integer	file descriptor
5	duplicate a file descriptor	integer	file descriptor
6	exit	integer	-
7	fork a new task	-	process id-old Ø-new
8	file status	integer, record	Ø - no error
9	get group identity	-	group id
1Ø	get process identity	-	process id
11	get user identity	-	user id
12	get tty status	integer, record	Ø-no error
13	kill	integer, integer	Ø-no error
14	link to a file	string, string	Ø-no error
15	nice priority	integer	
16	open a file	string, integer	file description

17	create a pipe	array	Ø-no error
18	read a file	integer, array, integer	Ø - endoffile >Ø - number of characters read
19	seek in a file	integer, integer, integer	Ø - no error
2Ø	set group identity	integer	Ø-no error
21	set user identity	integer	Ø-no error
22	catch or ignore signals	integer,integer	Ø-no error
23	sleep	integer	-
24	get file status	string,record	Ø-no error
25	set tty	integer, record	Ø-no error
26	sync	-	-
27	time	array	-
28	get process times	array	-
29	unlink directory entry	string	Ø-no error
3Ø	wait	-	status of last one
31	write to a file	integer,array,integer	Ø-no error

For more information, check Section II of the Unix Reference Manual. In addition, most of the above calls return a "-1" for errors.

1Ø. Sample Program

The following program is one solution to Dijkstra's "Dining Philosophers" problem[4]. We have five philosophers sitting in a circle with one fork between each philosopher. In the middle of the table is an unusual brand of spaghetti which requires two forks to eat. The philosophers must all cooperate in order to

eat. The program will print a "1" when a philosopher starts to eat; and will print a "2" when eating is over and thinking starts. The exported procedures "join" and "leave" are used to regulate eating habits. Note that since the data structures are modulated the philosophers cannot cheat to get at the food.

```
module dining;
        const NOTHINKERS=4;    (*number of philosophers*)
        var    cnt :integer;
        interface module table;
                define join,leave;
                use NOTHINKERS,cnt;
                var state:signal;
                    forks:array Ø:NOTHINKERS of boolean;
                procedure join(i:integer);
                        var j : integer;
                        begin
                        j:=(i+1)  mod (NOTHINKERS+1);
                        loop
                                (*wait for both forks*)
                                when forks[j] and forks[i] do
                                        forks[j]:=false; forks[i]:=false;
                                        (*mark them busy*)
                                        exit;
                                wait(state); (*wait for status change*)
                                end;
                        end join;
                procedure leave(i:integer);
                        begin
                        (*set forks to true to indicate available*)
                        forks[(i+1) mod (NOTHINKERS+1)]:=true;  send(state);
                        forks[i]:=true;  send(state);
                        end leave;
                begin
```

```
        (*initialize all forks*)
        cnt:=0;
        repeat forks[cnt]:=true; inc(cnt) until cnt > NOTHINKERS;
        end table;
procedure eat;
        begin end eat;
process philosopher(a:integer);
        begin
        loop
                (*get forks; eat; release forks; think*)
                join(a); sys(1,a,1); eat; sys(1,a,2);  leave(a);
                end;
        end philosopher;
begin
        cnt:=0;
        (*create philosopher processes*)
        while cnt <= NOTHINKERS do philosopher(cnt); inc(cnt) end;
        loop eat; end;
end dining.
```

LIST OF RESERVED KEYWORDS

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

NOT RESERVED

addr	dec	ord
array	false	send
awaited	halt	signal
bits	inc	sys
boolean	integer	true
char	nil	wait

NAME

mod - Modula compiler

SYNOPSIS

mod [-c] [-o] [-v] [-l] [-P] file ...

DESCRIPTION

Mod is the UNIX Modula compiler. It accepts three types of arguments:

Arguments whose names end with `.m' are taken to be Modula source programs; they are compiled, and each object program is left on the file whose name is that of the source with `.o' substituted for `.m'. The `.o' file is normally deleted if a single Modula program is compiled and loaded all at one go.

The following flags are interpreted by mod. See ld (I) for load-time flags.

-o Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-c Compile the named Modula source programs, but do not produce object files. This option is useful when removing compile-time errors.

-l Instruct the compiler to send a listing of the named source programs to standard output.

-P Instruct the compiler to generate a listing of the source program after C preprocessing. Note that if the first character of the source file is a #, the C preprocessor is automatically invoked.

-v Verbose. Writes to standard output what mod is doing.

Other arguments are taken to be either loader flag arguments, or Modula-compatible object programs, typically produced by an earlier mod run, or perhaps libraries of Modula-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name a.out.

FILES

file.m	input file
file.o	object file
file.i	preprocessed file
a.out	loaded output
temp.r	temporary
/usr/lib/modula/modula	compiler -- pass 1
/usr/lib/modula/mxa.out	compiler -- pass 2
/usr/lib/modula/mxload	runtime support routines
/usr/lib/modula/errors	compiler error messages

SEE ALSO

"An Introduction to Modular Programming for Pascal Users", db (I), ld (I).

DIAGNOSTICS

The diagnostics produced by Modula itself are intended to be self-explanatory. Occasional messages may be produced by the loader.

[1] N. Wirth, 'Modula: A language for modular multiprogramming', Software--Practice and Experience, V7, No. 1, 3-35(1977).

[2] N. Wirth, 'The use of Modula', Software--Practice and Experience, V7, No. 1, 37-65(1977).

[3] N. Wirth, 'Design and implementation of Modula', Software--Practice and Experience, V7, No. 1, 67-84(1977).

[4] E.W. Dijkstra, 'Hierarchical ordering of sequential processes', Acta Informatica, V1, No. 2, 115-138(1971).