An experimental implementation of the programming language modula.

John William Iobst

Follow this and additional works at: http://preserve.lehigh.edu/etd
Part of the Computer Sciences Commons

Recommended Citation
AN EXPERIMENTAL IMPLEMENTATION OF
THE PROGRAMMING LANGUAGE MODULA

by

John William Iobst

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy of the Degree of
Master of Science
in
Computer Science

Lehigh University
1979
This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Sept. 17, 1979
(date)

------------------
Professor in Charge

------------------
Chairman of Department
ACKNOWLEDGEMENTS

The author wishes to thank Professor Samuel L. Gulden for his leadership, support, and advice. His reading of this thesis is also deeply appreciated.

I would like to give special thanks to Richard J. Cichelli for all that he taught me about computing and for his help in the preparation of this thesis.

Special thanks go to my parents, family, and friends, especially Suzann Thomas, for all their help and encouragement without which this project may never have been completed. I am also indebted to Janice Ealer for typing the manuscript.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>1</td>
</tr>
<tr>
<td>I. DESIGN OF MODULA</td>
<td>3</td>
</tr>
<tr>
<td>II. FUNCTION OF MODULA</td>
<td>9</td>
</tr>
<tr>
<td>III. BASIC MODULES OF THE COMPILER</td>
<td>15</td>
</tr>
<tr>
<td>IV. CODE GENERATION FOR MODULA</td>
<td>33</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>40</td>
</tr>
<tr>
<td>APPENDIX 1: BNF FOR MODULA</td>
<td>42</td>
</tr>
<tr>
<td>APPENDIX 2: CODE INSTRUCTION MNEMONICS</td>
<td>48</td>
</tr>
<tr>
<td>APPENDIX 3: SAMPLE PROGRAM WITH GENERATED CODE</td>
<td>51</td>
</tr>
<tr>
<td>APPENDIX 4: VITA</td>
<td>57</td>
</tr>
<tr>
<td>FIGURE 1: Dependence Diagram For Modula</td>
<td>16</td>
</tr>
<tr>
<td>CODE SEGMENT 1: Module</td>
<td>17</td>
</tr>
<tr>
<td>CODE SEGMENT 2: Block</td>
<td>20</td>
</tr>
<tr>
<td>CODE SEGMENT 3: Body</td>
<td>22</td>
</tr>
<tr>
<td>CODE SEGMENT 4: Statement</td>
<td>25</td>
</tr>
<tr>
<td>CODE SEGMENT 5: Call</td>
<td>27</td>
</tr>
<tr>
<td>CODE SEGMENT 6: Assignment</td>
<td>28</td>
</tr>
</tbody>
</table>
ABSTRACT

Modula is a computer language developed at Stanford University. Professor Niklaus Wirth directed the language design and implementation group. The language was designed to be small in size, so that it would fit on small computers. In this sense the language design is minimal, while including all necessary features for concurrent programming and modern data structuring. It was anticipated that Modula would be used for the development of process control and other dedicated computer systems. In the future it is likely that Modula or one of its derivatives will be used as an implementation language for more general computer systems.

Modula is a direct descendant of Pascal. As such, many of the basic control and data structures come directly from Pascal. Although Modula is neither a subset or a superset of Pascal, it is a language of approximately equivalent power and it is intended to facilitate programming concurrent processes.

The compiler, which was developed, uses an LL(1) grammar. This grammar eliminates the use of parse tables.
and allows the compilation process to proceed by recursive descent. This technique is fast and efficient and will support error recovery in a clean manner. Details of the compilation process are presented along with the complete LL(1) grammar for the language.
I. DESIGN OF MODULA
---------------------

The intent of Modula[1,2,3] is to provide a high level language which will easily support concurrent processing. It should not be surprising to note that the bulk of the language consists of facilities that are typical of many sequential programming languages. The new technology which Modula contains is in the area of support for the programming of concurrent processes. The concurrent facilities deal with the creation and synchronization of these processes.

Sequential Modula, at first glance, looks almost identical to Pascal[4]. The similarity is due in part to the fact that many of the data and control structures are copied in design, if not in form, from Pascal. The structure and type of most of the data structures is directly from Pascal. There are, however, additions and deletions. The design of the control structures comes from Pascal, although the form is modified. Modula has fewer control structures than Pascal, but they are generally more powerful.

The basic data types: integer, char, and Boolean appear in Modula. User defined scalar types are available
and are specified by enumeration. Data structuring is accomplished using the array and record concepts. Variant records are not supported. The new data type bits, a short Boolean array, are also introduced. It serves as a partial replacement for the set type. It is somewhat limited in its utility due to size restrictions. The number of entities addressable is restricted to the number of bits in the computer word. The primary use of bits is to facilitate easy modification of device registers.

Modula has five basic control structures. They are: IF, CASE, WHILE, REPEAT, and LOOP statements. All control statements except REPEAT require an END to terminate the statement sequence. REPEAT uses an UNTIL in the same manner as Pascal. ELSE and ELSIF serve as interim terminators in the more complex IF statement. The syntax for each of the control structures is elaborated in Appendix 1.

The IF statement is a refined extension of the Pascal IF statement. The intermediate clause "ELSIF <expression> then <statement sequence>" allows repeated testing of an expression(s), without the structural depth which a similar coding in Pascal would imply. There is no limit to the number of ELSIF clauses.
The CASE statement is similar to its Pascal counterpart. The only difference being a required BEGIN-END pair for each case. The WHILE statement and the REPEAT statement are the same as Pascal's, with the exception of the required END for a WHILE.

The most unique control structure in Modula is the LOOP statement. The LOOP statement is the most general structure for support of the repeated execution of statements. REPEAT, WHILE, and Pascal's FOR statements are all special cases of the LOOP. The basic difficulty with the LOOP construct is that it will allow termination in the middle, as well as the beginning and the end of the construct. This makes program verification more difficult. Modula will also support multiple exits from a LOOP statement.

Modula and Pascal have many similarities. It is where Modula diverges from Pascal that the true power of Modula becomes evident. The added power shows in both the data and code segments of the language.

The critical addition to the data area is the data type bits. As previously mentioned, the data type bits allow the testing and setting of specific bits in a word. This ability, along with the ability to declare a variable to be
at an address, allows a variable of type bits to function as a device status register. The device and buffer registers for a terminal with the base address of 175640(8) would be:

```
var
readbufferstatus[175640B]: bits; (*output buffer status*)
readbuffer[175642B]: bits; (*output character buffer*)
writebufferstatus[174644B]: bits; (*input buffer status*)
writebuffer[175646B]: bits; (*input character buffer*)
```

To write a character to the terminal the transmit ready bit in the writestatusbuffer is checked. If the bit is set, the character to be output is placed in the writebuffer. The read operation will find a character in the readbuffer after an interrupt with a receiver interrupt enable bit set.

The previous example is for a terminal interface for a PDP-11. On a CDC 6000 series machine, a data structure, which represents the peripheral processor (PP) word in the task header, could be generated and set to the appropriate address. This would allow user programs to make PP calls.

An important addition to the control structures is not in functionality, but in form. Unlike Pascal where the IF construct is:
IF <expression> THEN <statement>

The Modula equivalent is:

IF <expression> THEN <statement sequence> END

It is much easier to use the second form than the first. The second form allows the programmer to add or delete statements with impunity. To add an additional statement to the Pascal IF-statement requires the addition of not only the new statement but also a BEGIN-END pair to bracket the statement sequence. Modula requires a terminator for all control structures. The terminator is an END, except for the REPEAT which uses an UNTIL to terminate.

Modula was designed as both a process control and systems implementation language. As such, it is necessary for Modula to support the creation and interprocess communication of concurrent processes.

To support the processes, the concept of a module was created. The basic module is similar to Hansens process [6,7]. An interface module will act to exclude two or more co-operating processes from operating on a common data item(s) at the same time. It accomplishes this by
restricting access to the data to the requesting process with the highest priority. Tasks of equal priority receive access on a first come, first serve basis. A higher priority task will interrupt a task of lower priority for access to common data. When a process has been completed, the highest priority process requesting access will be initiated. An interface module is similar to Hansen's [6,7] and Hoare's monitor [8].

A process in Modula is similar to a procedure. The difference is that, while a procedure executes to the exclusion of its calling program, a process executes concurrently with its calling program. A process may be synchronized with its parent program and other processes by means of signals. Signals are declared like variables but may have no value. They may only be sent or waited upon.
II. FUNCTION OF MODULA

Modula is designed to be equally functional as either a sequential or a concurrent processing language. The sequential aspect of Modula is much like Pascal. The concurrent processing portion of Modula is where the true power of the language becomes apparent.

The basic building block for a concurrent process is a module. A module definition serves as an encapsulation. This allows the control of access to names, items, and the existence of variables when their scope of definition is not active. The interface module is a special type of module which controls simultaneous access to common objects from more than one process. If two processes attempt to access an interface module at the same time, the second is delayed until the first is completed or is waiting on an event.

A second building block for concurrent execution is the process. A process is a sequential algorithm which is to execute concurrently with other processes. The only requirement for process speed is that it must be greater than zero.
A standard means of communication is established by the use of signals. A process is allowed to wait on a signal or a number of them. Signals may also be sent to processes which may be waiting for them. A process which is in an interface module and is waiting for a signal may release the module for access by other processes. If the original process is reactivated while another process is in the interface module, the first process regains control of the interface module even though the current process is not complete.

Modules, processes, and signals are all intricately entwined to allow Modula to support concurrent processing. Modules and processes provide a means to implement the algorithms used for concurrent processing. Signals provide a means of communication and synchronization between concurrent processes.

In order to make concurrent execution possible, it is necessary for procedures to have controlled access to data and possibly code that is owned by a different procedure. Access is granted or requested by the name definition concepts of use and define. Use allows the names of data and code items to be available to a routine other than that
which owns them. Define serves the opposite function by allowing only those names which are contained in the define list to be made available to a non-name owning routine.

Modula requires strict checking of the data and program names which are imported via the use list. The checking is strict to the extreme that only imported procedures may operate on imported data.

Example:

Two co-operating routines - one is a producer of data, the other is a consumer. The two routines communicate through the procedures p and c respectively. The flow of the data is from the producing routine, produce, through the interface procedures p and q, via the data buffer buff, to the consuming routine consume.

produce -> p -> buff -> c -> consume

interface module passbuffer;
define p, q, buff;
type
  buff = record
    size : 1..bufflen;
    data : array[1..bufflen] of dataitem
  end;
var
fillbuffer, bufferfull : signal;

procedure p(var buffer : buff);
begin
  wait (fillbuffer);
  send (bufferfull)
end p;

procedure c(var buffer : buff);
begin
  wait (bufferfull);
  if buffer.size > 0 then send (fillbuffer)
end
end c;

procedure cinit(var buffer : buff);
begin
  send (fillbuffer)
end cinit;
begin
end passbuffer;

process produce;
use p, buff;
var
  complete : boolean;
begin
  complete := false;
  while not complete do
    generate data (buff);
    p (buff)
  end;
  buff.size := 0;
  p (buff)
end produce;

process consume;
use c, cinit, buff;
var
  moredata : boolean;
  buff1, buff2 : buff;
begin
  cinit (buff1);
c (buff1);
moredata := buff1.size > 0;
while moredata do
  processdata (buff1);
c (buff2);
moredata := buff2.size > 0;
if moredata then
  processdata (buff2)
end
end consume;

This example shows two routines which pass data through an interface module. The producing routine uses a single buffer while the consuming routine is double buffered. This scheme may be used to even out speed differences between the two routines.

The consume routine has initial control of the data transfer. The call to cinit initiates the passage of data from p to c. The first call to c returns the first buffer of data in the variable buff1. Succeeding buffers of data are returned in buff2 then buff1 until all of the data has been transferred. Procedure c will always try to get a new buffer transferred until an empty buffer is sent.

The produce routine calls p when it has produced a buffer full of data. The routine then remains idle until a
fillbuffer signal is received. If p is waiting with a buffer full of data, the data is transferred when the signal fillbuffer is sent. If p is not active, i.e. waiting on a signal, then c remains active but waiting until p is activated.
III. BASIC MODULES OF THE COMPILER

The technique of parsing by recursive descent is used as the basic form for the compiler. The grammar for the language Modula is LL(1). Recursive descent[5] is one of the simplest and most effective ways of parsing a grammar of this type. The flow of the compiler, as it parses a program, follows the syntax diagram for the language. Each node in the syntax is a new procedure in the compiler. The dependence graph for Modula appears in Figure 1.
Figure 1
procedure module(fsys: setofsys);

var
  lcp, nxt: ctp;
  proiname: alfa;
  definechain, usechain, chainentry: chainvar;
  modulehead: boolean;
  oldlev: 0 .. maxlevel;
  oldtop: disprange;
  labname: integer;
  llc: addrrange;

begin (*module*)
  modulehead := true;  llc := lc;  dp := true;
  lc := lcaftermarkstack;  oldlev := level;
  oldtop := top;
  if level < maxlevel then level := level + 1
else error(251);
  if top < displimit then
    begin
      top := top + 1;
      with display[top] do
        begin
          fname := nil;  flabel := nil;  occur := blck
        end
    end
else error(250);
  new(usechain);
  with usechain do
    begin
      name := ';  nextvar := nil end;
  usechain .level := level;
  new(definechain);
  with definechain do
    begin
      name := ';
      nextvar := nil end;
  definechain .level := level;
  if not (sy in [interfacesy, modulesy, devicesy])
then begin error(6);  skip(fsys) end
else
  begin
    if sy in [interfacesy, modulesy] then
      begin
        if sy = interfacesy then begin insymbol; end;
        if sy <> modulesy then error(3)
      else
        begin
          insymbol;
          if sy <> ident
          - 17 -
then 
  begin 
  proname := ' '; error(2); 
  end 
else proname := id; 
insymbol; 
  if sy <> semicolon then error(14) 
end 
end else 
begin (*device module*) 
insymbol; 
  if sy <> modulesy then error(3) 
else 
  begin 
  insymbol; 
  if sy <> ident then 
  begin 
  proname := ' '; error(2) 
  end 
else proname := id; 
insymbol; 
  if sy <> lbrack then error(11) 
else 
  begin 
  searchsection(display[top].fname, lcp); 
new(lcp, mods, declared, actual); 
with lcp do 
  begin 
  name := id; accessway := all; 
idtype := nil; pflev := level; 
genlabel(labname); 
pfdeckind := declared; 
pfkind := actual; pfpri := 0; 
pfname := labname 
end; 
with lcp do 
  begin 
  if inchain(name, definechain) 
  then accessway := define; 
if inchain(name, usechain) then 
  if accessway = all 
  then accessway := use 
  else error(101) 
end; 
if lcp .accessway = all 
then enterid(lcp);
if lcp.accessway = define
then enterdefine(lcp, definechain);
insymbol;
if sy <> intconst then error(15)
else
begin
insymbol;
if sy <> rbrack then error(12)
else
begin
insymbol;
if sy <> semicolon
then error(14)
end
end
end
end
end;
repeat
insymbol;
if sy = definesy
then
(* set up for exportable names *)
definename;
if sy = usesy
then
(* set up for imported names *)
usename;
until not (sy in [definesy, usesy, semicolon, comma]);
if top = 1 then dummyproc;
repeat block(fsys, period, nil, progname)
until (sy = period) or (id = progname);
level := oldlev; top := oldtop; lc := llc
end (*module*);

Code Segment 1: Module
------------------------

Following the initialization phase of the compiler, the first routine that is called is MODULE. This routine is shown in Code Segment 1 and has two functions. The first is - 19 -
to ascertain the type of module which is being processed. The types of modules being: interface, device, or others. The interface module defines a program section which encapsulates routines which handle critical sections of a program, i.e. those areas which require critical exclusion of processes. A device module is a portion of a program which contains one or more machine dependent sections of code. The third type of module, designated "other", is used for program definition and as a means of controlling access to named elements. The definition of a module creates a "fence" around its data and procedures. The creation of this fence is the second function of a module. The operators Use and Define serve as paths through which objects may be passed into or out of a fenced area. This allows compile time checking of object accessibility, along with the potential of objects continuing to exist even after the termination of their scope of definition.

procedure block(fsups: setofsys; fsy: symbol; fprocp: ctp; endid: alfa);

var
  lsy: symbol;

begin (*block*)
  repeat
    if sy = constsy
    then begin insymbol; constdeclaration end;
    if sy = typesy

- 20 -
then begin insymbol;   typedeclaration end;
if sy = varsy
then begin insymbol;   vardeclaration end;
while sy in [interfacesy, moduleysy, devicesy] do
  begin
    module(fsys);
    if sy <> semicolon then
      begin error(6);   skip(fsys + [semicolon]) end;
    insymbol
  end;
while sy in [proceduresy, processsy] do
  begin
    lsy := sy;   insymbol;
    proceduredeclaration(lsy);
  end
until sy in (statbegsys + [valuesy]);
if not (sy in [beginsy, valuesy])
then begin error(6);   error(17);   skip(fsys) end;
if sy = valuesy
then begin insymbol;   initialvalues end;
testl(beginsy, 17);
repeat
  body(fsys + [casesy]);
  if (sy <> fsy) and (id <> endid)
  then begin error(6);   skip(fsys) end
  until (sy = fsy) or (sy in blockbegsys) or (id = endid);
end (*block*);

Code Segment 2:  Block
----------------------

After the module heading is processed, the next routine which is called is BLOCK and is shown in Code Segment 2. BLOCK processes all of the declarations which are local to one section of a program. Unlike Pascal, where the order of declarations is defined as one pass, the declaration section for Modula is cyclic in nature. This allows, for example, constants to be declared after types, and variables to be
declared before types. The only requirement is that the declaration of an object must occur before its use. When the declaration processing is complete, BLOCK calls BODY.

procedure body(fsyp: setofsys);

const
cixmax = 1000;
cstoccmax = 65;

type
  oprange = 0 .. 63;

var
cstptr: array [1 .. cstoccmax] of csp;
cstptrix: 0 .. cstoccmax;
lcmax, ldisp, llc1: addrrange;
i, entname, segsize: integer;
stacktop, topnew, topmax: integer;
lcp: ctp;
llp: lbp;

begin (*body*)
  dp := false;
  if fprocp <> nil then entname := fprocp .pfname
  else genlabel(entname);
  cstptrix := 0;  topnew := lcaftermarkstack;
  topmax := lcaftermarkstack;  putlabel(entname);
  genlabel(segsize);  genlabel(stacktop);
  gencupent(32, 1, segsize);  gencupent(32, 2, stacktop);
  if fprocp <> nil
  then (* copy multiple values into local cells*)
     begin
        llc1 := lcaftermarkstack;  lcp := fprocp .next;
        while lcp <> nil do
          with lcp do
            begin
              align(parmptr, llc1);
              if klass = vars
                then
                  if idtype <> nil
                    then
                      if idtype .form > subrange

- 22 -
then
begin
if vkind = actual then
begin
  gen2(50 (*lda*), 0, vaddr);
  gen2t(54 (*lod*), 0, llc1, nilptr);
  gen1(40 (*mov*), idtype .size);
end;
llc1 := llc1 + ptrsize
end
else llc1 := llc1 + idtype .size;
lcp := lcp .next
end;
lcmax := lc;
repeat (*loop until sy <> semicolon*)
  statement(fsys + [semicolon, endsy]);
  if sy in statbegsys then error(14)
until not (sy in statbegsys);
while sy = _semicolon do
begin
  insymbol;
  repeat
    statement(fsys + [semicolon, endsy]);
    if sy in statbegsys then error(14)
  until not (sy in statbegsys)
end;
test1(endsy, 13);
if fprocp <> nil then
begin
  gen0t(42 (*ret*), fprocp .idtype);
  align(parmptr, lcmax);
  writeln(prr, '1', segsize: 4, '=' , lcmax - lcaftermarkstack);
  writeln(prr, '1', stacktop: 4, '=' , topmax)
end
else
begin
  gen1(42 (*ret*), 0);  align(parmptr, lcmax);
  writeln(prr, '1', segsize: 4, '=' , lcmax);
  writeln(prr, '1', stacktop: 4, '=' , topmax);
  writeln(prr, 'q');   ic := 0;  dp := true;
(* generate call of main program; note that this call
must be loaded at absolute address zero *)
gen1(41 (*mst*), 0);
gencupent(46 (*cup*), 0, entname);  gen0(29 (*stp*));
writeln(prr, 'q');
The procedure BODY is the first routine in the compilation process which will generate code. The code for this routine is shown in Code Segment 3. The first code which BODY generates is the code necessary for the entry point to the routine and the generation of the new stack frame. After the new stack frame is generated, a copy of any parameter values is placed into local cells. At this point, the procedure STATEMENT is called repeatedly until the current symbol is no longer in the set of statement begin symbols. This concludes the statement processing within the routine BODY. All that is left for BODY at this point is to generate the code for a routine termination and to re-align the stack.
procedure statement(fsys: setofsys);

label
1;

var
lastsy: symbol;
lcp: ctp;

begin (*statement*)
if not (sy in fsys + [ident])
then begin error(6);   skip(fsys) end;
if sy in statbegsys + [ident]
then
begin
    lastsy := sy;
    if sy = ident
    then
        begin
            searchid([vars, field, proc, mods], lcp);
            testaccess(usechain, lcp);   insymbol;
            if lcp .klass in [proc, pros, mods]
            then call(fsys, lcp)
            else assignment(fsys)
        end
    else
    begin
        insymbol;
        case lastsy of
            beginsy: compoundstatement(fsys);
            ifsy: ifstatement;
            casesy: casestatement;
            whilesy: whilestatement;
            repeatsy: repeatstatement;
            loopsy: loopstatement;
            withsy: withstatement
            end (*case*)
        end;
    test2(fsys, 6, []);
    end;
end (*statement*);

Code Segment 4: Statement
The procedure STATEMENT processes all of the information having to do with statements, as defined in the syntax for the language. The code for STATEMENT appears in Code Segment 4. If the current symbol is an identifier, the symbol table is searched to determine the class which is associated with the identifier. At this time, the accessibility of the identifier is also checked. An error will be recorded if the identifier is not accessible to this module. If the class of the identifier is procedure, process, or module, then the routine CALL is executed. If the symbol is not an identifier, then a case statement is executed and the appropriate control statement is determined and processed. The control statements are: a compound statement which is defined by the symbol BEGIN, an if statement which is defined by the symbol IF, a case statement which is defined by the symbol CASE, a while statement which is defined by the symbol WHILE, a repeat statement which is defined by the symbol REPEAT, a loop statement which is defined by the symbol LOOP, and a with symbol which is defined by the symbol WITH.
procedure call(fsys: setofsys; fcp: ctp);

var
  lkey: 1 .. 25;
  access: accesstype;
  uselevel: disprange;

begin (*call*)
  access := fcp .accessway;
  if inchain(fcp .name, usechain)
    then getuselevel(fcp .name, usechain, uselevel);
  if fcp .pfdeckind = standard
    then
      begin
        lkey := fcp .key;
        if fcp .klass = proc
          then
            if lkey <> 3
              then
                begin (*standard procedures*)
                  test1(lparen, 9);
                  case lkey of
                    1: inc;
                    2: dec;
                    4: wait;
                    5: send;
                    6: awaited
                  end;
                  test1(rparen, 4)
                end
              else halt
          else
            begin (*standard functions*)
              test1(lparen, 9);  expression(fsys + [rparen]);
              load;
              case lkey of
                7: off;
                8: among;
                9: low;
                10: high; 11: adr;
                12: size;
                13: ordf;
                14: chr
              end;
              test1(rparen, 4)
            end
        else
          end
      else
        else
          -27-
begin (*nonstandard procedures and functions*)
callnonstandard
end
end (*call*);

Code Segment 5: Call
----------------------

The procedure CALL processes all subprogram calls. If the subprogram to be executed is a standard procedure or function, the appropriate standard call is made. Code Segment 5 contains the code for CALL. All other procedures and functions go through the procedure CALLNONSTANDARD, which checks to assure the appropriate parameters are specified. If it is a function call, space is allocated on the stack for the return value of the function.

procedure assignment(fcp: ctp);
var
  lattr: attr;
  lcix1, lcix2: integer;
begin
  selector(fsys + [becomes], fcp);
  if sy = becomes
  then
    begin
      if gattr.typr <> nil then
        if (gattr.access <> drct) or (gattr.typr .form > subrange)
          then loadaddress;
      lattr := gattr;
      (* save attributes of storage point *)
      insymbol; expression(fsys);
      if gattr.typr <> nil then

    - 28 -
if gattr.ty PTR .form <= subrange then load
else load address;
if (lattr.ty PTR <> nil) and (gattr.ty PTR <> nil)
then
begin
if comtypes(lattr.ty PTR, gattr.ty PTR)
then
begin
if comptypes0attr . ty PTR, gattr.ty PTR)
then
begin
if debug then checkbnds(lattr.ty PTR);
if lattr.ty PTR = bitptr
then
begin
  genlabel(lcix1); genfjp(lcix1);
genO(68 (*sbt*)); genlabel(lcix2);
genupxjp(57 (*ujp*), lcix2);
putlabel(lcix1); gen0(67 (*cbt*));
putlabel(lcix2)
end; store(lattr)
end;
arrays, records:
gen1(40 (*mov*), lattr.ty PTR .size)
end
else error(129)
end
end (*sy = becomes*)
else error(51);
end (*assignment*);

Code Segment 6: Assignment
---------------------------

The procedure ASSIGNMENT processes all of the
assignment statements in a Modula program. Assignment is
shown in Code Segment 6. ASSIGNMENT calls SELECTOR to
generate the list of attributes about the target variable.
The attributes vary depending on whether the variable is a
simple type, an array, a field, or a function. The
necessary code is generated to load the address of the target of the assignment, if that target is not directly addressable, or if its form is not simple. At this point, the procedure EXPRESSION is executed to process the right side of the assignment. When EXPRESSION is complete, the value on the top of the stack is stored in the target of the assignment.

EXPRESSION leaves the result of its evaluation on the top of the stack. The appropriate code is then generated to store the result and, if desired, range checking is performed. EXPRESSION analysis in Modula uses the same precedence order as Pascal. The first operation is to process a simple expression. The simple expression consists of that part of an expression which is only divisible by relational operators. If a relational operator is encountered after the simple expression is processed, the appropriate code is indicated for the relation and a second simple expression is processed.

The procedure SIMPLEEXPRESSION, in conjunction with its local procedures TERM and FACTOR, processes all of the arithmetic operations in an expression, along with the Boolean operations AND, OR, and NOT. SIMPLEEXPRESSION also
enforces the arithmetic precedence rules for expression evaluation. At each level, the variables to be operated on are loaded onto the stack. The appropriate operation is performed and the result left on top of the stack. Once all of the operations are performed, which are required to evaluate the expression, the final result of the expression is left on top of the stack. This is true whether the expression is to be used in an assignment or if the expression is used as part of a relational operator.

The routine COMPOUNDSTATEMENT is a special case of a routine STATEMENT. It groups a series of statements together as one logical entity which is surrounded by a BEGIN END in the simple case or one of the reserved words in a controlled structure in the general case.

The control structures for Modula are the IF statement, CASE statement, REPEAT statement, WHILE statement, and the LOOP statement. Each of these is processed by its own special routine which generates the appropriate jumps and calls to COMPOUNDSTATEMENT. It is interesting to note in the IF statement, that the construct ELSIF is supported. This addition tends to minimize the level of nesting of IF statements. The REPEAT statement and the WHILE statement
are identical to those which are used in Pascal, with the exception that the WHILE statement does not require a BEGIN after the DO to indicate a compound statement. A WHILE statement always terminates with an END symbol. Modula has replaced the FOR statement from Pascal with LOOP statement which is the general case for all iterative statements. The LOOP statement could, in fact, be used as a replacement both for the REPEAT and WHILE statements, as they are special cases.
IV. CODE GENERATION FOR MODULA

The basic form of the code which the Modula compiler generates comes from the Pascal P compiler system [9]. The code is called P-code. A list of the available instructions appears in Appendix 2. The generated code is for a hypothetical stack machine with no registers.

All operations are performed with reference to the data stack. Therefore, only two types of instructions, LOAD and STORE, are necessary to reference memory locations. Suitable derivatives of the instructions are provided to cover all necessary memory addressing, including the immediate loading of constants and memory addresses.

Two more basic types of instructions are needed. These instructions are classed as stack operations and program counter operations. Stack operators manipulate the top value or top two values of the data stack. A stack operation always leaves a value on the stack. A jump instruction modifies the value of the program counter. The jump instruction may use the value on top of the stack (TOS), a boolean value, for a conditional jump. Unconditional jumps leave the stack unchanged. A special
The case of the jump instruction is the procedure call.

The following describes the style of the code generated for the statements used in MODULA. A detailed example of a working program is presented in Appendix 3.

The code which would be generated for the assignment, $A := B + C$, would be of the form:

- LOAD $B$ on stack
- LOAD $C$ on stack
- ADD $B + C$
- STORE $A$ from stack

The code generated for a REPEAT-UNTIL construct such as REPEAT <statement sequence> UNTIL <expression> would be:

L1:
- code for <statement sequence>
- code for <expression>
- JUMP to L1 if TOS is false

The value on the top of the stack after the expression is evaluated is always the result of the evaluation. This
allows the use of a jump-false instruction which may be used after the evaluation of an expression in any of the control structures.

The code generated for the WHILE-DO statement of the type:

```
WHILE <expression> DO
  <statement sequence>
END
```

would be:

```
L1:
  code for <expression>
  JUMP to L2 if TOS is false
  code for <statement sequence>
  JUMP unconditional to L1
L2:
```

This construct uses the same jump-false instruction as the REPEAT.

The code generated for an IF-THEN statement such as:
IF <expression>(1) THEN
  <statement sequence>(1)
ELSIF <expression>(2) THEN
  <statement sequence>(2)
ELSE
  <statement sequence>(3)
END

would be:

code for <expression>(1)
JUMP to L1 if TOS is false
code for <statement sequence>(1)
JUMP unconditional to L3
L1:
  code for <expression>(2)
  JUMP to L2 if TOS is false
code for <statement sequence>(2)
  JUMP unconditional to L3
L2:
  code for <statement sequence>(3)
L3:

The code for a LOOP statement such as:

LOOP
  <statement sequence>(1)

- 36 -
WHEN <expression>
DO <statement sequence>(2)
EXIT
<statement sequence>(3)
END

L2:
L1:
code for <statement sequence>(1)
code for <expression>
JUMP to L2 if TOS is false
code for <statement sequence>(2)
JUMP unconditional to L3
code for <statement sequence>(3)
JUMP unconditional to L1

The last control structure is the case statement. The generated code for a case statement is different from all of the other control structures. The efficient implementation of a case statement requires the use of a jump table to generate the appropriate section of code for execution. In order to create this table, the compiler must maintain a
list of all of the case labels encountered and their locations in the code stream. After the contents of the case statement have been processed the case table is generated. There is one entry in the table for each case label, with any missing labels being replaced by jumps to a run-time error report. Neither the "else" nor "otherwise" construct is supported. The code for the case statement:

```
CASE I of
  1:
  BEGIN
    <statement sequence>(1)
  END;
  2:
  BEGIN
    <statement sequence>(2)
  END;
  4:
  BEGIN
    <statement sequence>(4)
  END
END
```

would be:
JUMP to case table L5

L1:
    code for <statement sequence>(1)
    JUMP Unconditional to L6

L2:
    code for <statement sequence>(2)
    JUMP Unconditional to L6

L4:
    code for <statement sequence>(4)
    JUMP Unconditional to L6

L5:
    JUMP Indexed based on ordinal of case variable
    JUMP to L1 for I=1
    JUMP to L2 for I=2
    JUMP to ERROR for I=3
    JUMP to L4 for I=4

L6:
REFERENCES


APPENDIX 1

BNF stands for Backus-Naur Form. This notation allows a simple, uniform method of specifying the syntax of MODULA.

The special symbols {} and | are used. {B} indicates that the symbol B is used zero or more times. A|B indicates that either A or B is the symbol to be used.

BNF for MODULA

<program> ::= <module>.

<module> ::= <module heading> <define list> <use list> <block> <indent>

<module heading> ::= interface module <ident>; | module <ident>; | device module <ident> <priority>;

<define list> ::= define <ident list>; | <empty>

<use list> ::= use <ident list>; | <empty>

<block> ::= {[<declaration part>]} <initialization part> <statement part> end

<declaration part> ::= const <constant declaration>; {[<constant declaration>];} | type <type declaration>; {[<type declaration>];} | var <variable declaration>; {[<variable declaration>];} | <module>; | <procedure declaration>
; | <process declaration>;

<ident list> ::= <ident> {, <ident>}

<constant declaration> ::= <ident> = <constant>

<type declaration> ::= <ident> = <type>

<ident> ::= <letter> [<letter or digit>]

<letter or digit> ::= <letter> | <digit>

<constant> ::= <unsigned constant> | <sign> <integer>

<unsigned constant> ::= <ident> i <integer> |
    | '<character>
    | {<character>}
    | <octal digit>
    | [<constant>]
    | [,<constant>]
    | [,<subrange>]
    | [,<subrange>]

<sign> ::= + | -

<integer> ::= <digit> {<digit>} | <octal digit>
    | {<octal digit}B

<subrange> ::= <constant> : <constant>

<type> ::= <ident> | [<ident> {,<ident>}] |
    | array <subrange> [{,<subrange}] of type | record <fieldlist> end

<fieldlist> ::= <record section> {;<record section>}

<record section> ::= <ident> {,<ident>} : <type> |
    | <empty>

<variable declaration> ::= <ident> {,<ident>} : <type>

<procedure declaration> ::= <procedure heading> <block>
    | <ident>

<procedure heading> ::= procedure <ident>; | <ident>
    | procedure <ident>
    | <formal parameters>|
    | procedure <ident> : <ident>;
    | <ident>
    | <formal parameters>:

- 43 -
procedure <ident>; <uselist>
procedure <ident>
<formal parameters>;
<uselist>
procedure <ident>: <ident>;
<uselist>
procedure <ident>
<formal parameters>:
<ident>; <uselist>

<formal parameters> ::= <section> {;<section>}

<section> ::= var <parameter group> |
        const <parameter group>
              <parameter group>

<parameter group> ::= <ident> {,<ident>} :
       <formal type>

<formal type> ::= <ident> | array <index types> of <ident>

<index types> ::= <ident list>

<process declaration> ::= <process heading>
       <uselist> <block> <ident>

<process heading> ::= process <ident>; |
       process <ident>
       <formal parameters> ; |
       process <ident> <intvector> ; |
       process <ident>
       <formal parameters>
       <intvector>;

<intvector> ::= [<integer>]

<initialization part> ::= value <ident> =
       <initial value>
       {<ident> = <initial value>}

<initial value> ::= <constant> | [<repetition>]
       <initial value> |
       (<initial value>
       {,<initial value>})

<repetition> ::= <integer> | <ident>
<statement part> ::= begin <statement sequence>

<statement sequence> ::= <statement> {;<statement>}

<statement> ::= <assignment> | <procedure call> | <process statement> | <if statement> | <case statement> | <while statement> | <repeat statement> | <with statement>

<assignment> ::= <variable> := <expression>

<procedure call> ::= <ident> | <ident> <parameter list>

<parameter list> ::= (<parameter> {,<parameter>})

<parameter> ::= <expression> | <variable>

<process statement> ::= <ident> | <ident> <parameter list>

<if statement> ::= if <expression> then <statement sequence> <elsif part> <else part> end

<elsif part> ::= elsif <expression> then <statement sequence> <elsif part> | <empty>

<else part> ::= else <statement sequence> | <empty>

<case statement> ::= case <expression> of <case list element> {; <case list element>} end

<case list element> ::= <constant> {,<constant>} : begin <statement sequence> end

<while statement> ::= while <expression> do <statement sequence> end

<repeat statement> ::= repeat <statement sequence> until <expression>
<loop statement> ::= loop <statement sequence>
<when part> end

<when part> ::= when <expression> exit
<statement sequence>
<when part> | when <expression> do
<statement sequence> exit
<statement sequence>
<when part> | <empty>

<with statement> ::= with <variable> do
<statement sequence>
end

<variable> ::= <entire variable> | <component variable>

<entire variable> ::= <ident>

<component variable> ::= <indexed variable> | <field designator>

<indexed variable> ::= <array variable> [<expression>
{,<expression>} | <bit variable>
[<expression>]

(array variable) ::= <variable>

(bit variable) ::= <variable>

(field designator) ::= <record variable> .
<field identifier>

<record variable> ::= <variable>

(field identifier) ::= <ident>

(expression) ::= <simple expression> |
<simple expression>
<relational operator>
<simple expression>

(relational operator) ::= = | <> | <= | < | > | >=

(simple expression) ::= <term> | <sign> <term> |
<simple expression>
<adding operator>
<term>
<adding operator> ::= + | - | or | xor

<term> ::= <factor> | <term> <multiplying operator> <factor>

<multiplying operator> ::= * | / | div | mod | and

<factor> ::= <unsigned constant> | <variable> |
          <function designator> | (<expression>)
          | not <factor>

<function designator> ::= <ident> | <ident>
           <parameter list>
## APPENDIX 2

### P-Code Instruction Mnemonics

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>produce absolute value of integer</td>
</tr>
<tr>
<td>ADI</td>
<td>produce sum of integers</td>
</tr>
<tr>
<td>ADR</td>
<td>address of variable passed</td>
</tr>
<tr>
<td>AMG</td>
<td>bit specified true</td>
</tr>
<tr>
<td>AND</td>
<td>perform Boolean 'and'</td>
</tr>
<tr>
<td>AWT</td>
<td>waiting on signal</td>
</tr>
<tr>
<td>CBT</td>
<td>clear bit specified</td>
</tr>
<tr>
<td>CHK</td>
<td>check that the top of stack is in range</td>
</tr>
<tr>
<td>CHR</td>
<td>convert integer to character</td>
</tr>
<tr>
<td>CSP</td>
<td>call standard procedure</td>
</tr>
<tr>
<td>CUP</td>
<td>call user procedure</td>
</tr>
<tr>
<td>DEC</td>
<td>decrement top of stack by amount</td>
</tr>
<tr>
<td>DVI</td>
<td>integer divide</td>
</tr>
<tr>
<td>ENT</td>
<td>enter block</td>
</tr>
<tr>
<td>EQU</td>
<td>test for equality</td>
</tr>
<tr>
<td>FJP</td>
<td>jump if stack top false to label</td>
</tr>
<tr>
<td>GEQ</td>
<td>test for greater than or equal to</td>
</tr>
<tr>
<td>GRT</td>
<td>test for greater than</td>
</tr>
<tr>
<td>HGH</td>
<td>high index bound of array passed</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>INC</td>
<td>increment top of stack by amount</td>
</tr>
<tr>
<td>IND</td>
<td>indexed fetch</td>
</tr>
<tr>
<td>IOR</td>
<td>perform Boolean 'inclusive or'</td>
</tr>
<tr>
<td>IXA</td>
<td>compute indexed address</td>
</tr>
<tr>
<td>LAO</td>
<td>load base-level address</td>
</tr>
<tr>
<td>LCA</td>
<td>load address of constant</td>
</tr>
<tr>
<td>LLA</td>
<td>load address</td>
</tr>
<tr>
<td>LDC</td>
<td>load constant</td>
</tr>
<tr>
<td>LDO</td>
<td>load contents of base-level address (global variable)</td>
</tr>
<tr>
<td>LEQ</td>
<td>test for less than or equal to</td>
</tr>
<tr>
<td>LES</td>
<td>test for less than</td>
</tr>
<tr>
<td>LOD</td>
<td>load contents of address</td>
</tr>
<tr>
<td>LOW</td>
<td>low index bound of array passed</td>
</tr>
<tr>
<td>MOD</td>
<td>modulus</td>
</tr>
<tr>
<td>MOV</td>
<td>moves the number of storage units given</td>
</tr>
<tr>
<td>MPI</td>
<td>multiply integers</td>
</tr>
<tr>
<td>NEQ</td>
<td>test for not equal</td>
</tr>
<tr>
<td>NGI</td>
<td>negate integer</td>
</tr>
<tr>
<td>NOT</td>
<td>perform Boolean not</td>
</tr>
<tr>
<td>OFF</td>
<td>bit set empty</td>
</tr>
<tr>
<td>ODD</td>
<td>test for odd</td>
</tr>
<tr>
<td>ORD</td>
<td>convert to integer</td>
</tr>
<tr>
<td>RET</td>
<td>return from block</td>
</tr>
</tbody>
</table>
SBI  perform integer subtraction
SBT  set bit specified
SGS  generate singleton set
SQI  square integer
SRO  store at base-level address
STO  store indirect
STP  stop
STR  store
TBT  test bit specified
UJC  error in case statement-abort
UJP  unconditional jump to label
given by Q
UNI  perform union of sets
XJP  indexed jump; jump to offset
    + top of stack
Sample Program

MODULE LINEINPUT;
 DEFINE READ, NEWLINE, NEWFILE, EOLN, EOF, LNO;
 USE INCHR, OUTCHR;
 CONST LF = 12C; CR = 15C; FS = 34C;
 VAR
   LNO: INTEGER; (*LINE NUMBER*)
   CH: CHAR; (*LAST CHARACTER READ*)
   EOF, EOLN: BOOLEAN;

PROCEDURE NEWFILE;
 BEGIN
   IF NOT EOF THEN
     REPEAT INCHR(CH) UNTIL CH = FS;
   END;
   EOF := FALSE; LNO := 0
 END NEWFILE;

PROCEDURE NEWLINE;
 BEGIN
   IF NOT EOLN THEN
     REPEAT INCHR(CH) UNTIL CH = LF;
     OUTCHR(CH);OUTCHR(LF)
   END;
   EOLN := FALSE; LNO := LNO + 1
 END NEWLINE;

PROCEDURE READ(VAR X: CHAR);
 BEGIN
   LOOP INCHR(CH);OUTCHR(CH);
     WHEN CH >= ' ' DO X := CH EXIT
     WHEN CH = LF DO X := ' '; EOLN := TRUE EXIT
     WHEN CH = FS DO X := ' '; EOLN := TRUE; EOF := TRUE EXIT
   END
 END READ;
 BEGIN
   EOF := TRUE; EOLN := TRUE
 END LINEINPUT.
Generated Code for Sample Program

L5 (*entry to Newfile*)

ENT 1 L6
ENT 2 L7
LDOB 8 (*load value in EOF*)
NOT (*negate TOS*)
FJP L8 (*jump if TOS false*)

L9

LAO 6 (*load address of ch*)
CUP 1 L3 (*call Inchr*)
LDOC 6 (*load contents of ch*)
LDCC 'FS' (*load constant 'FS'*)
EQUC (*test top 2 values on stack for equality*)
FJP L9 (*jump if TOS false*)

L8

LDCB 0 (*load boolean constant 'false')
SROB 8 (*store boolean TOS EOF*)
LDCI 0 (*load integer constant 0*)
SROI 5 (*store integer TOS at LNO*)
RET (*return*)

L6 = 0 (*segment and stack maximum size*)
L7 = 8

- 52 -
L10 (*entry to Newline*)

ENT  1 L11
ENT  2 L12
LDOB  7 (*load value in EOLN*)
NOT (*negate TOS*)
FJP L13 (*jump if TOS false*)

L14

LAO  6 (*load address of ch*)
CUP  1 L3 (*call Inchr*)
LDOC  6 (*load value in CH*)
LDCC 'CR' (*load constant 'CR'*)
EQUC (*test top 2 values on stack for equality*)
FJP L14 (*jump if TOS false*)
LAO  6 (*load address of CH*)
CUP  1 L4 (*call Outchr*)
LDCC 'CR' (*load constant 'CR'*)
CUP  1 L4 (*call Outchr*)

L13

LDCB  0 (*load constant 'false'*)
SROB  7 (*store boolean TOS in EOLN*)
LDOI  5 (*load value of LNO*)
LDCI  1 (*load constant 1*)
ADI (*integer add of top 2 values on stack*)

- 53 -
SROI 5 (*store integer TOS in LNO*)
RET (*return*)
L11 = 0 (*segment and stack maximum size*)
L12 = 10
L15 (*entry to Read*)
ENT 1 L16
ENT 2 L17

L18
LAO 6 (*load address of CH*)
CUP 1 L3 (*call Inchr*)
LAO 6 (*load address of CH*)
CUP 1 L4 (*call Outchr*)
LDOC 6 (*load value of CH*)
LDCC '' (*load constant ' '*)
GEQC (*test top 2 values on stack for greater than or equal*)
FJP L20 (*jump if TOS false*)
LODA 0 5 (*load address of X*)
LDOC 6 (*load value of CH*)
STOC (*store TOS in X*)
UJP L19 (*jump to label*)

L20
LDOC 6 (*load value of CH*)
LDCC 'LF' (*load constant 'LF'*)
EQUC (*test top 2 values of stack for equality*)
FJP L21 (*jump if TOS false*)
LODA 0 5 (*load address of X*)
LDCC ' ' (*load constant ' '*)
STOC (*store TOS in X*)
LDCB 1 (*load boolean constant 'true'*)
SROB 7 (*store TOS in EOLN*)
UJP L19 (*jump to label*)

L21
LDOC 6 (*load value of CH*)
LDCC 'FS' (*load constant 'FS'*)
EQUC (*test top 2 values on stack for equality*)
FJP L22 (*jump if TOS false*)
LODA 0 5 (*load address of X*)
LDCC ' ' (*load constant ' '*)
STOC (*store TOS in X*)
LDCB 1 (*load boolean constant 'true'*)
SROB 7 (*store TOS in EOLN*)
LDCB 1 (*load boolean constant 'true'*)
SROB 8 (*store TOS in EOF*)
UJP L19 (*jump to label*)

L22
UJP L18 (*jump to label*)

- 55 -
L19
   RET  (*return*)
L16    =  1  (*segment and stack maximum size*)
L17    =  19
L23
   ENT   1   L24
   ENT   2   L25
   LDCB   1  (*load boolean constant 'true'*)
   SROB   8  (*store TOS in EOF*)
   LDCB   1  (*load boolean constant 'true'*)
   SROB   7  (*store TOS in EOLN*)
   RET    0  (*return to monitor*)
   STP    (*stop*)
John William Iobst was born in Allentown, Pennsylvania to Ralph William and Grace Marie (Baugh) Iobst on December 31, 1953. He grew up in Emmaus, Pennsylvania and attended Emmaus High School. A Bachelor of Science degree in Chemistry was earned from Moravian College in Bethlehem, Pennsylvania in 1975.