Chas. Irby

MPS Language Reference Manual

4 SEP 72

MPS 11.0

James G. Mitchell*

Xerox Palo Alto Research Center*
3180 Porter Drive
Palo Alto, CA 94304
(415) 493-1600

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 326-6200

MPS Language Reference Manual
Mitchell
SRI/XPARC

MPS 11.0
4 SEP 72
PAGE 2

.ID -- recognizes a lower case identifier,

.UID -- recognizes an upper case identifier,

.NUM -- recognizes a number,

.SR -- recognizes a string enclosed in quotes ("),

.SR1 -- recognizes a single character

preceded by an apostrophe ('), or

.CHR -- recognizes any character;

a string enclosed in quotes (");

a single character string indicated by an apostrophe (') followed by the character;

a list of alternatives enclosed in parentheses;

a dollar sign ($) followed by an element, which means an arbitrary number of occurences (including zero) of the element. The notation $(x)y is used as an abbreviation for y $(x y).

Comments enlosed in percent signs (%) may be embedded anywhere in the rule. The rule is terminated by a semicolon (;). .PES;

MODULE

```
module =
 label
 ("DATA" [varnames] ';
   [directory] $declare /
 "PROGRAM" [varnames] ';
   [directory] $declare programbody)
 "END.";
```

```
label = '( .ID ');

varnames = '( $<',>formal ');

  formal =

   ["INTEGER"] .ID ["SIZE" constexp] /
   "STRING" .ID /
   "SIGNAL" .ID /
   "FIELD" .ID /
   "PROCEDURE" .ID /
   "ARRAY" .ID;

programbody = $procedure $blkdcl $<';>labeled;
```

An MPL source file is either a DATA module or a PROGRAM module.

DATA modules are to be used to hold declarations of shared data structures, global parameters which are referenced from several locations, commonly used macro definitions, etc.

The DATA module defines a storage area which may be accessed relative to a base pointer. Many instances of the same module may be created. (See the CREATE and BASE statements.)

The parameters of the DATA module can be used in expressions to determine the dimensions of arrays or the initial values of variables when an instance of the module is created (loaded).

PROGRAM modules will be used to hold code and declarations which are strictly local to that code. It will thus be possible to edit and recompile a PROGRAM module without affecting other modules as long as the module's interfaces remain the same.

The run-time equivalent of the PROGRAM module is the "process".

A process consists of an instance of the storage declared for the PROGRAM module and the compiled code for the body and procedures of the module.

There may be several distinct processes created from a single PROGRAM module.

The PROGRAM module also may have parameters which can be used in the initialization of process data. The values of these parameters are saved in the process data segment and are thus available throughout the life of the process.

(NOTE: These are NOT parallel processes or independant processes in the eyes of the time-sharing system. All the MPL-processes exist in a single TSS-process.)

Directory

```
directory = "DIRECTORY" $(',)(label ["INCLUDE"] link) ';;

  link = '( dir ', file ', name ');
```

The directory at the front of a module has two uses; it establishes default bindings for external procedure references, and it specifies COMMON modules whose declarations are to be known in this module.

A directory entry such as

```
(outproc) (directory, file, name)
```

informs the system (and anyone reading the module) that "outproc" will be used as the name of an external procedure and that the intended binding is to a procedure to be found at the location specified by the link.

It is possible to reference an external procedure without having listed it in the directory. The directory simply provides a means of specifying a default binding.

It is also possible to "override" a default binding and redirect the references to that procedure to another one.

A directory entry such as

```
(othermod) INCLUDE (directory, file, name)
```

indicates that "othermod" may be used as the name of a module whose location is given by the link, and that the declarations in that module are to be known in this one at compile time.

Procedure

```
procedure =

  label "PROCEDURE" [varnames] ';

  $declare  $blkdcl  $(';)labeled  "END.";
```

A procedure may have an arbitrary number of local variables, which include formal parameters, variables, arrays, strings, etc.

Note that the dimensions and initial values of local quantities are allowed to depend on the parameters of the procedure.

Scopes

Every declaration of a name in MPL affects interpretation of that name within a well-defined scope.

The scope of a declaration in a procedure body is the rest of that procedure, starting immediately after the ', or '; that ends the declaration.

The scope of a declaration in the module head is the rest of the module.

When scopes overlap, the interpretation of a name follows a priority order:

current procedure

current module

INCLUDEd modules, in reverse order

Since a variable name stands only for a position within an instance of its declaring scope, there are rules for determining the base address for variables declared in each type of scope.

Procedure variables are based on the current activation of the procedure.

Program variables are based on the data segment for the particular instance of the process.

DATA module variables may be declared to be based on a particular pointer with the statement

"BASED" modulename "ON" .ID ';

Otherwise their variables must be accessed with explicit field references using the unary '/ operator described below.

DECLARATION FORMS

Declare

declare = (decl / record / register) ';;

```
decl = "DECLARE" (

 ["INTEGER"] $(',)item /
 "STRING" $(',)strdcl /
 "SIGNAL" sigdcl /
 "ARRAY" $(',)ardcl /
 "FIELD" $(',)fdef /
 "PROCEDURE" procv /
 "PORT" $(',)prtdef;

item = .ID ['← exp / '= constexp / size];
```

An identifier by itself serves to declare a simple integer variable.

If the identifier is followed by a left arrow (←) and an expression, then whenever an instance of the variable is created, the expression is evaluated and the result stored into the variable to initialize it.

```
 constexp = exp; %which can be evaluated at compile time%
```

If the identifier is followed by an equals (=) and a constexp (compile-time constant expression), then within the scope of the declaration, the name is equivalent to the value of the constant expression. (The identifier is NOT a variable and may be used only in those contexts in which a constant could appear.)

```
 size = "SIZE" constexp;
```

Until a more general data structure facility materializes, this syntax allows definition of a scalar variable of any number of words. Such variables are restricted to the following contexts:

On the left hand side of an assignment in which the right hand side is a fullword variable or a function call;

Assume that the multiword scalar on the left side of the assignment has been declared to be n words long. If the right side is a variable, then n words are copied starting from the location of the object on the right. If the right side is a function call, the first n words of results of the function are copied into the multiword.

On the right of an assignment in which the left hand side is a multiword;

As an extra result of a function call (see explanation of multiple results);

As an argument of a function call;

As an argument of "RETURN".

```
strdcl = .ID ['← exp / '[ exp ']];
```

An identifier by itself serves to declare a string variable.

If the identifier is followed by '← exp, then the value of the expression, which must be a string, is used to initialize the variable.

If the identifier is followed by '[exp '], then when an instance of the variable is created room is immediately reserved for a string of up to exp characters. (NOTE: the string may still grow beyond this bound and additional space will be allocated -- the exp is simply a way to control the initial allocation.)

```
sigdcl = "CODE" $(',).ID / $(',)(.ID ['← exp]);
```

SIGNAL CODES are given unique internal names and are "constants" (i.e. cannot be written by normal programs). These may be declared only in the body of a module (i.e. not in a procedure).

(NOTE: a cell is allocated to hold the signal code; during initialization of the data segment, the cell is set to its own segmented address).

If "CODE" is not present, then the identifiers are signal variables which may be initialized to the value of an expression.

```
ardcl = .ID ['[ exp '] / '← '( $(',)exp ')];
```

The identifer is declared as an array. If it stands alone, then it is an array variable and must be set to point to a real array before it is used.

If the identifier is followed by '[exp '], then room for exp entries is allocated -- these are indexed by integers in the range 0 to exp-1.

If the identifier is followed by ← (e1, ... en), then an array of n entries is allocated with the entries initialized to the corresponding exp.

```
fdef = .ID ['= '[ [offset ',] size ': position ']];
```

An identifier by itself indicates that this is a field variable.

```
offset = constexp;
```

The offset is the word displacement of the field relative to the origin of the record. Nonzero offsets are used when extracting a field from a multiword record.

```
size = constexp;
```

This is the size in bits of the field.

```
position = constexp;
```

This is the position of the field in the word given as a displacement in bits from the right.

```
procv = "PROCEDURE" $(',)(.ID ['← exp]);
```

The listed identifiers contain procedure descriptors.

Procedure variables may be used as an argument of a function call. They may be set ONLY in contexts which accept a multiword scalar. The procedure variable may be initialized.

```
port = "PORT" $(',).ID;
```

The listed identifiers may be used as port names. This declaration may occur only in the head of a PROGRAM module.

Record

```
record = label "RECORD" $(',)recdef;

recdef = .ID ('[ constexp '] / '( $(',)recdef '));
```

The RECORD declaration defines a group of fields of specified sizes: the compiler assigns the actual bit positions starting a the right end of relative word 0 and working to the left.

A single field must not exceed one word in size.

The second alternative allows a contiguous group of fields to be given a name.

The entire group must fit in one word.

Note that at present, the field names are not "local" to the record in any way and therefore must not duplicate other defined names in the same scope.

Register

```
register = "REGISTER" $(',)regdef;

regdef = .ID ['= constexp];
```

The constexp specifies the actual address to be used to hold the variable.

Addresses less than 20B on the PDP-10 refer to the machine registers.

If no constexp is given, then the compiler allocates a register. This register will be taken out of the pool used for expression evaluation. The contents of the register is NOT saved over procedure/port/etc. calls, so use with caution.

VARIABLE FORMS

The rule "lhs" (left hand side) gives the syntax for the variable forms. These variables may be used on the left hand side of assignment statements as well as in expressions and certain other syntactic positions.

```
lhs = (fwlhs / '( exp ') '. field) $qualifier;

fwlhs = % full word left-hand-side %

scalar / arrayelt / contents;
```

Scalar variables

```
scalar =

RegisterVariable /

ProcedureVariable /

ProgramVariable /

DataVariable;
```

A scalar variable may be a register or it may be local to a procedure, a process, or an instance of a DATA module.

RegisterVariable

```
RegisterVariable = .ID;  %declared to be a REGISTER%
```

ProcedureVariable

```
ProcedureVariable = .ID;  %formal parameter or local variable%
```

Procedure variables include formal parameters and variables declared at the head of a procedure. Each time a procedure is called, space is allocated for its variables. The space is released when the procedure returns. Thus procedures may be used recursively and each activation of the procedure will have a private set of variables.

ProgramVariable

```
ProgramVariable = .ID;
```

Program variables are those declared at the beginning of the PROGRAM module. Each time a process is created, storage is allocated for its variables. This space is released when the process is destroyed. In addition, the program variables can be saved and restored by means of the PUSH and POP ENVIRONMENT statements (see below).

DataVariable

```
DataVariable = .ID;
```

Data variables are those declared in DATA modules. The user must create and destroy instances of them explicitly.

Array elements

```
arrayelt = ArrayName '[ exp '];

ArrayName = .ID;  %which was declared to be an array%
```

Arrays are indexed from 0.

Contents variables

```
contents = '[ exp '];
```

Contents variables refer to the location whose address equals the value of the expression.

In the simplest case where the expression is a variable, indirect addressing is used. More complex expressions are evaluated and then indexing is used.

Qualifier

```
qualifier = ('$ / '.) field;

field =

 .ID  %declared to be a field% /

 '( exp ') %value used as field descriptor% /

 '/ .ID;
```

To reference a declared variable relative to a base pointer, one must obtain a field which selects that variable from the record containing it. The unary '/ operator does this, e.g. to reference an INCLUDEd variable x relative to a base pointer p one writes p./x. (See also the BASE declaration).

Otherwise, the identifier used as a "fieldname" must have occurred in either a FIELD or RECORD declaration.

A field descriptor consists of the size, position, and address displacement of the field. The size is the number of bits in the field and the position is the number of bits in the word to the right of the field.

Any variable form may be qualified by following it by a dollar sign and a fieldname. This results in accessing the contents of the named field within that variable. It is possible to refer to a field within a field. For example, to zero the c field of the b field of the variable k the statement

```
k$b$c ← 0
```

may be used.

If the variable form is followed by a period and a fieldname, the field is selected in the object pointed to by value of the variable form, i.e. a.b is equivalent to [a]$b.

A more general data definition facility is planned for a future version of MPL.

STATEMENT FORMS

```
labeled = $label stat;

stat =
```

```
assign / conditional / dostatement / SimpleControl / ProcessControl / SignalStatements
/ MiscStatements;

assign = lhs '← exp;
```

The expression is evaluated and then stored into the left hand side.

```
conditional = if / case;
```

There are two types of conditional statements: the common "IF" statement and a "CASE" statement.

```
 if = "IF" exp "THEN" labeled ["ELSE" labeled];

 case = "CASE" sum

  ("OF" $cofst / "IN" $cinst / ropr $copst)

  "ENDCASE" labeled;

   cofst = $<',>(binrel / sum) ': labeled ';;

   cinst = $<',>(intrel / charclass) ': labeled ';;

   copst = $<',>sum ': labeled ';;
```

The CASE-statement provides a means of executing one statement out of many. The sum after the word "CASE" is evaluated and the result saved. This is used as the left-hand side of the relations at the beginning of the various cases. A sum used at the head of a case is taken to mean =sum. Several relations may be listed at the start of a single statement; the statement will be executed if any of the relations are satisfied. If none of the relations are satisfied, the statement following the word "ENDCASE" will be executed.

Example:

```
  CASE c OF

    a: x ← y;                  %c = a%

    b: x ← y-x;                %c = b%

    >b: x ← x-y;               %c > b%

    ENDCASE y ← x;             %c # a AND c < b%
```

If a case ends with an unconditional transfer, the compiler does not produce another (unnecessary) branch instruction.

dostatement = $doclause "DO" labeled [dotest] [thenclause];

The DO statement provides for controlled iteration. The statement following the DO is repeatedly executed until either a dotest fails, the controlled variable reaches its bound, or control leaves the DO statement because of some action by the controlled statement (such as RETURN, EXIT, or other transfer statement).

thenclause = "THEN" labeled;

If the iteration is terminated because a dotest fails or the controlled variable reaches its bound, the statement in the thenclause is executed before control leaves the DO statement. However, if the DO statement is exited by other means, then the thenclause statement is not executed.

doclause = dotest / dovarclause;

dotest = "WHILE" exp / "UNTIL" exp;

A dotest of the form WHILE exp succeeds if the exp evaluates to TRUE. The dotest UNTIL exp succeeds if the exp evaluates to FALSE. There may be a single dotest at the head of the DO statement and another at the tail. The test at the head is made before each execution of the controlled statement and the test at the tail is made after each execution.

The remaining types of doclauses provide for the specification, initialization, stepping, and testing of the controlled variable of the DO statement. (In the following, the controlled variable is referred to as "CV").

" "FOR" lhs ['← exp] [', exp] /

The lhs is used as the CV.

If the "← exp" is present, the exp is used to initialize the lhs. Otherwise, if there is no initial value specifed by another dovarclause, then the value of the lhs is left unchanged.

If the ", exp" is present, then after each iteration the expression is evaluated

and stored into the CV. This offers an alternative to simply incrementing (or decrementing) the CV and is particularly useful for following lists.

Example:

FOR x, x.next TO Nil DO ...

The DO statement is terminated if the CV equals the TO exp.

If no FOR clause appears, then a cell is automatically allocated to hold the CV, and if no initial value is specified by another dovarclause, then the cell is set to zero.

"FROM" exp /

The exp is used to initialize the CV.

("UP" / "DOWN") [exp] /

The exp is evaluated and saved as the increment for the CV. The default value for the exp is 1. If no clause of this form is given, then the default direction is up.

If the direction is up, then after each execution of the controlled statement (and any trailing dotest) the CV has the increment added to it. Similarly, if the direction is down, then the increment is subtracted from the CV.

"TO" exp /

The exp is evaluated and saved as the bound for the CV.

After the controlled statement has been executed and the CV has been assigned a new value, the CV is compared to the saved value of the bound. If there is a ", exp" in a FOR clause, then the DO statement is terminated if the CV equals the TO exp. Otherwise, if the direction is up and the CV is greater than the bound, then the DO statement is terminated. Similarly, if the direction is down and the CV is less than the bound the iteration stops.

"THRU" ('(/ '[) exp ', exp (') / ']);

The CV is stepped thru the interval.

If the direction is up, then the CV is initialized to the smallest integer in the interval and the bound is set to the largest integer in the interval.

If the direction is down, then the CV is initialized to the largest integer in the interval and the bound is set to the smallest integer in the interval.

```
SimpleControl = functionstatements / branchstatements;

  functionstatements = bind / call / return;

    bind = "BIND" ...
```

This statement is used to bind procedure variables and external procedure references to actual procedures.

```
    call = "CALL" fwlhs [args] / fwlhs args;
```

There are two forms of the procedure call statement, one starting with the word "CALL" and optional arguments, and the other without the word "CALL" and mandatory argument list.

```
     args = '( [$<',>exp] [': $<',>lhs] ['! ctchp] ');
```

The argument list begins with an arbitrary number of expressions separated by commas. The argument expressions are evaluated in order from left to right.

Following the arguments there may be a list of locations for extra results to be returned. The list of left hand sides for these multiple results is separated from the list of argument expressions by a colon. The number of locations for results need not equal the number of results actually returned. If there are more locations than results, then the extra locations get an undefined value. If there are more results than locations, the extra results are simply lost.

Example:

If procedure p ends with the statement

```
  RETURN (a, b, c)
```

then the statement

```
  q ← p( : r, s)
```

results in

```
  q ← a; r ← b; s ← c;
```

Finally, there may be a catch phrase to handle signals generated as a result of the call (see the discussion of catch phrases and signals below).

```
return = "RETURN" ['( $<',>exp ')];
```

A procedure may return an arbitrary number of results. The order of evaluation of results is from right to left.

```
branchstatements = "EXIT" labid / "REPEAT" labid;

labid = .ID; %which is a label%
```

The branch statements must refer to a label of an enclosing statement. In other words, a label can be used only within the statement which it preceeds.

EXIT label means terminate the execution of the labeled statement. Control is transferred as if the statement had terminated in the normal manner.

REPEAT label means restart the execution of the labeled statement.

```
ProcessControl =

 create / destroy / run / stop / portstat / environment;

  create = "CREATE" .ID ['( arglist ')] ["FROM" .ID];

  destroy = "DESTROY" .ID;

  run = "RUN" '( processname ['! ctchp] ');

  stop = "STOP" ['( [processname] ['! ctchp] ')];

  portstat = join / useport / empty;

    join = "JOIN" ...

    useport = "PORT" '( portname [', exp] ['! ctchp] ');

      portname = lhs; %which specifies a port%

    empty = "EMPTY" '( portname ['! ctchp] ');

    recentp = "RECENT" "PORT";

  environment = ("PUSH" / "POP") "ENVIRONMENT";

SignalStatements = signal / signalport / signalprocess / resume;
```

```
    signal =
     ("SIGNAL" / "ERROR") '( siglist ');
    signalport =
     ("SIGNAL" / "ERROR") "PORT" '( portname ', siglist ');
    signalprocess =
     ("SIGNAL" / "ERROR") "PROCESS" '( processname ', siglist ');
    siglist = code [', message] ['! ctchp];
    resume = "RESUME" ['( [exp] ['! ctchp] ')];
  MiscStatements = block / bump / syscall / inline / null;
    block = "BEGIN"  $blkdcl  $(';)labeled  "END";
```

The block statement simply allows the grouping of several statements into one.

The hierarchical structure of the NLS file will eventually replace the use of BEGIN's and END's as a means of forming compound statements.

```
     blkdcl = "ENABLE" ctchp / "BASE" $(',)based;
```

Catch phrases

```
       ctchp = $(ctchs ';);
         ctchs = $(',)(signame / syssig) ': labeled;
         signame = .ID;  %a SIGNAL variable or a SIGNAL CODE%
         syssig = "UNWIND" / "PORTFAULT" / ...
```

Based

```
       based = modulename "ON" .ID;
```

The modulename must correspond to a name occurring in the directory. The identifier must be the name of a simple variable declared in this procedure or program. For the scope of the BASE declaration, all uses of variable names from the specified module will automatically be made relative to the contents of the named variable.

```
bump = "BUMP" ["DOWN"] $(',)lhs;
```

The BUMP statement increments each of the variables, while the BUMP DOWN statement decrements each one.

```
syscall = "JSYS" '( constexp [', Jsysargs] $JsysResultOptions ');
```

This statement for the PDP-10 implementaion allows the user to call the TENEX monitor. The JSYS number is given by the constexp.

```
Jsysargs = $(',)exp;
```

The expressions e1, e2,.. are loaded into registers 1, 2,.. respectively.

```
JsysResultOptions = ': $(',)[lhs];
```

In general, the JSYS returns to the calling location plus i, where i=1 for normal return, i=2 for skip return, etc. The number i defined in this manner is used to choose one of the JsysResultOptions and is also taken as the value of the JSYS statement when it is used in an expression.

For the chosen result option, the registers 1, 2,.. are stored into lhs1, lhs2,.. If there is no lhs given for a particular register then that result is simply discarded.

Example:

```
IF JSYS(gtjfn,flags,stringp:errcode:jfnword,stringp) = 1 THEN jsyserr(errcode);
```

The variables flags and stringp are loaded into registers 1 and 2, then a JSYS gtjfn is executed. If the JSYS returns to the first location following the call, then the new contents of register 1 is stored into errcode. The value of the JSYS call is 1 in this case, so the function jsyserr is called. If the JSYS returns to the calling location plus 2, then registers 1 and 2 are stored into the variables jfnword and stringp and the value of the JSYS call is 2.

```
inline = "INLINE" '( $(';)instruction ');
```

The instruction syntax given here is for the PDP-10 implementation.

```
instruction =

[label] %label for instruction%
```

```
    .ID %opcode%

    [instcon ',] %accumulator%

    ['@] %indirection%

    ['= instcon / .ID / .NUM] %address%

    ['( instcon ')]; %index%
```

The opcode identifier must be declared as a constant; the value is placed in the top nine bits (the opcode field) of the instruction.

If an identifier is used as the address, then appropriate indexing is provided. (This allows instructions to address global or local variables without knowing the base register used for these).

```
 instcon =

   .NUM /

   .ID %must be defined and either a constant or a REGISTER%;

null = ["NULL"];
```

The null statement is provided as a convenience to the programmer.

EXPRESSION FORMS

Expressions

```
exp =

 "IF" exp "THEN" exp "ELSE" exp /

 "CASE" sum

   ("OF" $cofxp / "IN" $cinxp / ropr $copxp)

   "ENDCASE" exp /

 disjunct;

  cofxp = $<',>(binrel / sum) ': exp ';;

  cinxp = $<',>(intrel / charclass) ': exp ';;
```

```
copxp = $(',)sum ': exp ';;
```

An expression involving logical operators may be used in the place of an arithmetic expression. It has the value 1 if true and the value 0 if false: however, the compiler gives a warning message since this is a common coding error. This message can be suppressed by using the BOOLEAN pseudo-function described below.

Example:

```
flag ← a > b OR x = y
```

is equivalent to

```
flag ← IF a > b OR x = y THEN 1 ELSE 0.
```

Likewise, an arithmetic expression may be used where a logical expression is expected. A zero value represents false, and a nonzero value represents true.

Example:

```
IF p() THEN a ELSE b
```

is equivalent to

```
IF p() NOT= 0 THEN a ELSE b.
```

Logical operators

```
disjunct = $("OR")conjunct;
```

True if any conjunct is. The conjuncts are evaluated, left to right, only until a true one is reached or all have been found false.

```
conjunct = $("AND")negation;
```

True if all negations are. The negations are evaluated, left to right, only until a false one is reached or all have been found true.

```
negation = "NOT" negation / relation;

relation = sum [binrel];

binrel = ["NOT"] (
 (")=" / "(=" / ') / '( / '= / '#) sum /
```

```
"IN" (charclass / intrel);

charclass =

"CH"  / %any character%

"ULD" / %uppercase letter or digit%

"LLD" / %lowercase letter or digit%

"LD"  / %lowercase or uppercase letter or digit%

"NLD" / %not a letter or digit%

"UL"  / %uppercase letter%

"LL"  / %lowercase letter%

"L"   / %lowercase or uppercase letter%

"D"   / %digit%

"PT"  / %printing character%

"NP"    %nonprinting character%;
```

This provides a simple way to test the common classes of characters.

Example:

```
  char IN LD
```

is true if the variable "char" contains a value which is a letter or a digit.

```
intrel = ('( / '[) sum ', sum (') / ']);
```

This provides for the common operation of testing whether the value of some expression lies within a particular interval. Each side of the interval may be "open" or "closed". In other words the value which determines the boundary of the interval may be included within the interval (by using square brackets) or excluded (by using parentheses). The IN relation means that the value is in the interval.

Example:

```
  fl(z) IN [1,100)
```

is the same as

```
        f1(z) >= 1 AND f1(z) < 100.
```

except that f1 is only called once.

Arithmetic operators

```
    sum = $< '+ / '- > prod;
```

The plus (+) and minus (-) represent addition and subtraction.

```
    prod = $< '* / '/ / "MOD" > bitor;
```

The star (*) and slash (/) represent multiplication and division.

The form a MOD b gives the remainder of a / b.

```
    bitor =  $<"BITOR" / "BITXOR"> bitand;
```

The form a BITOR b gives the logical or of a and b.

The form a BITXOR b gives the exclusive or of a and b.

```
    bitand = $<"BITAND">factor;
```

The form a BITAND b gives the logical and of a and b.

```
    factor = '- factor / '( exp ') $qualifier / prim;
```

Note that it is possible to extract a field from the value of a parenthesized expression.

Example:

The assignment statement

```
    x ← (b + c)$f
```

stores the f field from b + c into the variable x.

Primitives

```
    prim =

    lhs [args $qualifier / '← exp / ":=" exp] /

    '$ fwlhs /

    .SR /
```

```
"PROCEDURE" .ID /

"BOOLEAN" '( exp ') /

"DIV" '( exp ': lhs ') /

"RECORDSIZE" '( .ID ') /

"DISPLACEMENT" '( .ID ') /

"FIELDMAX" '( .ID ') /

rhsstat /

ctchprhs /

'/ .ID /

literal;

("MIN" / "MAX") '( $<',>exp ') ;
```

When a procedure call is used as a primitive, the value is that of the leftmost result returned by the procedure.

Two forms of assignments can be used as primitives.

The form a ← b has the effect of storing b into a and has the value of b as its value.

The form a :≡ b has the effect of storing b into a and has the old value of a as its value. (Where possible, an exchange with memory instruction is done rather than a store).

Example:

The statement

```
c ← b :≡ c
```

exchanges the contents of b and c.

The DIV pseudo-function allows both the quotient and the remainder of a division to be saved. The central connective in the expression must be binary '/. The value of the DIV is the quotient of the division; the remainder is stored into the lhs following the colon.

RECORDSIZE(recordname) is the number of words in an instance of that record type.

DISPLACEMENT(variablename) is the word displacement of the variable.

FIELDMAX(fieldname) is the maximum value that can be stored in the named field. This is limited to constant fields (i.e. FIELDMAX is a compile-time function).

The pseudo-functions MIN and MAX have the obvious meaning.

A full word left hand side preceded by a dollar sign ($) represents the address of that item.

A .SR yields the address of that string.

Procedure (descriptor) constants must be linked like external procedure references and therefore must be indicated explicitly.

```
rhsstat = signal / signalport / create / useport / empty;
```

These statements can produce a value and thus may be used as right hand sides.

```
ctchprhs = "CODE" / "MESSAGE";
```

Inside a catch phrase statement, these refer to the values of the signal code and message respectively.

The /ID construct yields a field descriptor for the named variable.

```
literal = .NUM / "TRUE" / "FALSE" / char;
```

Numbers come in several flavors. A sequence of digits alone or followed by a D is interpreted as base ten. If followed by a B then it is interpreted as base eight. A scale factor may be given after the B for octal numbers or after a D for decimal numbers. The scale factor is equivalent to adding that many zeros to the original number.

Examples:

64 = 100B = 1B2

144B = 100 = 1D2

A sequence of digits followed by an M is a mask. The number indicates the number of bits to be turned on in the mask. An octal scale factor may also be present to the right of the M. If the scale factor is absent, the mask is right justified. Thus

18M is an 18 bit mask on the right half of a 36-bit PDP-10 word; 18M6 is an 18 bit mask on the left half word.

The words TRUE and FALSE are equivalent to the numbers 1 and 0 respectively.

The following provide synonyms for commonly used characters. The effect is the same as if the number internally representing the character had been used.

```
char =

  .SRl  /  %single character preceded by an apostrophe%

  "CD"  /  %command delete%

  "SP"  /  %space%

  "EOL" /  %Tenex's version of CR LF%

  "ALT" /  %Tenex's version of altmode or escape (=33B)%

  "CR"  /  %carriage return%

  "LF"  /  %line feed%

  "TAB" /  %tab%

  "BC"  /  %backspace character%

  "BW"  /  %backspace word%

  "CDOT"  /  %center dot%

  ("ENDCHR" / "EOS") / %as returned by string system%

  ("BUG" / "CA") %command accept%;
```

CODE PRODUCED BY THE COMPILER

The following is provided as an admittedly contrived example of the code produced by the compiler.

For the sequence of statements

```
x ← x + a;

y ← 6 - y;
```

```
IF x < 0 THEN
  BEGIN
  z ← 1;
  [p] ← x;
  BUMP p;
  END
ELSE z ← -z;
[p] ← x;
a ← z;
```

the compiler produces the following instructions:

```
MOVE 17,a          % r17 ← a %
ADDB 17,x          % x ← r17 ← r17 + x %
MOVEI 16,6         % r16 ← 6 %
SUBB 16,y          % y ← r16 ← r16 - y %
JUMPGE 17,.+6      % if r17 (holding x) >= 0 then goto .+6 %
MOVEI 15,1         % r15 ← 1 %
MOVEM 15,z         % z ← r15 %
MOVEM 17,@p        % [p] ← r17 (which still holds x) %
AOS 14,p           % p ← r14 ← p + 1 %
JRST .+2           % branch over false part of IF statement %
MOVNS 15,z         % z ← r15 ← -z (use same register for z) %
MOVEM 17,@p        % [p] ← r17 (which still holds x) %
MOVEM 15,a         % a ← r15 (which holds z since was loaded
                     in both parts of the IF statement) %
```

The compiler remembers simple variables which are in accumulators and carries this information forward through the various paths of the IF statement. Notice that in both the true and false parts of the IF, the variable z is loaded into the same register so that in the following statements z need not be reloaded.