%MPL (B)%

%A Modular Programming Language for Software Construction%

This document describes a programming language, MPL(B), for use on MAXC and the PDP-10. The language attempts to strike a balance between efficiency, machine independence, and user convenience. MPL(B) is an interim language providing limited extensions of MPL(A) in a form intended to be compatible with subsequent versions of MPL.

E. SatterthWaite (editor)
Xerox Palo Alto Research Center
Palo Alto, Ca.

\ ph ← TRUE; NewPage(); pn ← TRUE; pageno ← 1; blm ← -ilev

The definition of the syntax is given by a set of productions, each of which consists of a left part, the sequence "::=", a right part, and a terminating semicolon (";").

The left part is a phrase class name constructed from lower case letters, digits, and hyphens.

The right part of a production consists of one or more alternatives. Multiple alternatives are separated by vertical bars ("_"). Each alternative consists of a sequence of elements. Any subsequence enclosed in square brackets ("[" and "]") is optional. All other elements in the sequence must occur in the specified order. The elements may be any of the following:

a phrase class name;

one of the following tokens, each representing a class of basic symbols:

.ID -- any identifier,

.NUM -- any number,

.SR -- any string, or

.SR1 -- any character constant;

one of the following literals, each of which denotes the quoted value:

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 occurrences (including zero) of the element;

a pound sign ("#") followed by an element, which means one or more occurrences of the element.

The following abbreviations are used, where x and y are arbitrary strings:

[x]	for	*[(x)]*
$<x>y	for	*[y $(x y)]*
#<x>y	for	y $(x y) .

\ blm ← -2*ilev

%BASIC SYNTACTIC ENTITIES%

An identifier (.ID) is a sequence of letters or digits which begins with a letter. Upper and lower case letters are distinct. Identifiers consisting only of upper case letters are reserved to denote basic symbols of MPL. Other identifiers have no inherent meaning but can be introduced to identify variables, types, constant values, etc.

Examples:

i BindByName X2 typeSTRING

Examples (reserved):

PROCEDURE FOR STRING

A number (.NUM) is a sequence of digits optionally followed by a "B" (indicating an octal number) or a "D" (indicating a decimal number) and a decimal scale factor. If neither letter appears, the number

is assumed to be decimal. The scale factor indicates the number of zeros to be appended to the original string of digits.

Examples (all numbers have the same value):

1000 1000D 1D3 1750B 175B1

A string (.SR) is a (possibly empty) sequence of characters enclosed within quotes. Within the string, the quotation character itself can be denoted by a doubled quotation character; otherwise, all characters in a string (including nonprinting and control characters) denote themselves.

Examples:

"a string" "" """"

A character constant (.SR1) consists of an apostrophe followed by a character.

Examples:

'a '" ' ''

Blanks and other separators (such as tab or end-of-line characters) cannot be embedded within numbers or identifiers. At least one such separator must appear between two adjacent identifiers or numbers. Except within strings, separators may otherwise be used freely; neither they nor the tree structure of the NLS file containing an MPL program has any inherent meaning. A comment is any sequence of characters beginning with an unbroken pair of dashes ("--") and ending with a similar pair of dashes or an EOL character (possibly the implicit EOL which terminates every NLS statement). A comment can appear wherever a separator is allowed.

%TYPES AND VALUES%

MPL(B) is a weakly typed language. Every variable has a well-defined %type%, which is specified when the variable is declared. The type determines how the values of the variable are to be represented, accessed, and manipulated. In particular, there is an implementation-dependent length associated with each type. The length is the number of bits allocated to allow the representation of any possible value of that type; it may exceed the number of bits required by the minimal representation.

The value of an expression is simply a sequence of bits. Thus an expression does not have a type but does have a length. The length of a constant is determined by the syntactic form of the constant itself. The length of the value of a variable is equal to the length associated with the type of that variable. Values can be combined by using a set of machine-defined operations upon bit strings with certain standard lengths. Most of these operations imply an interpretation of their operands, e.g., as binary

representations of integers. In addition, bit sequences can be interpreted as pointers used to access further bit sequences or as descriptors of composite objects, such as procedures or strings, with standard interpretations defined by the MPS support package.

If the length of a bit string is less than or equal to the machine's word length, that bit string is classified as a %uniword%; otherwise, as a %multiword%. More precisely, the value of a variable is classified as a uniword or a multiword according to the type of the variable. To avoid possible anomalies arising from quirks of implementation, the value of a variable with a record or array type is always classified as a multiword, although the implementation may choose to pack such a value into a single word. The operators of the language maintain the distinction between uniwords and multiwords consistently, even when such packing occurs.

In MPL(B), the length of a multiword is always an integral multiple of the word length.

A constant value which is a uniword can be written as an unsigned octal number. No denotations of multiword constants are provided.

On MAXC and the PDP-10, the word length is 36, and the longest sequence of one-bits which is a uniword is 777777777777B.

The value of an expression can be assigned to any variable having the proper length. Certain automatic conversions are allowed. Explicit conversion rules applicable in specific contexts appear in subsequent sections (see especially assignments and procedure calls). Whenever conversion is allowed but no such explicit rules are given, the following %default conversion rules% apply:

A uniword of any length can be converted to a uniword of any other given length, and the conversion is automatic. The bit sequence is extended (by prefixing zero bits) or truncated (by discarding leading bits) as required.

Note that there is no check for loss of sign or significance in these conversions.

No other automatic conversions are allowed, i.e., automatic conversions involving multiwords are disallowed.

MPL(B) does not check the consistency and compatibility of types. Variables should, however, be appropriately declared and consistently used, both for documentation and for compatibility with future versions of MPL, which will be more strongly typed.

%Basic Types%

A variable of type INTEGER stores a uniword. Integers are considered to be signed; the value of an integer variable is obtained by interpreting the bit sequence as a two's complement binary number. Integer constants are written as ordinary decimal or octal numbers.

On MAXC and the PDP-10, the range of an integer variable is -34359738368 to 34359738367.

An interval *[m .. n]* is the set of all integers i such that m <= i <= n. The range of the type IN *[m .. n]* , where m and n are nonnegative constants, is the set of integers in the specified interval. Any such type is a %set-element% (or, more simply, element) type. An element variable stores sequences of bits at least long enough to represent the value of the largest integer within the interval as an unsigned binary number. In the specification of an interval, an endpoint may be included (indicated by a square bracket) or excluded (indicated by a round bracket). An element value is a uniword; thus constants can be written as ordinary octal or decimal numbers.

Abbreviations are available for certain common set-element types, and there are alternative ways of writing some of the associated constants (see EXPRESSION FORMS).

\ Grab(7)

Abbreviation	Type
BOOLEAN	IN *[0..1]*
CHARACTER	IN *[0..177B]*
WORD	IN *[0..777777777777B]*

A variable of type FIELD contains a uniword describing a selector that can be applied to a bit sequence to extract a subsequence. The syntax of field constants and the representation of field values are discussed in connection with primaries (q.v.).

The syntax and semantics of FIELD values are likely to change in future versions of MPL.

An identifier of type TYPE is used to name the description of a programmer-defined type which, in MPL(B), must be a record type. These descriptions are constants; there is no provision for dynamic type evaluation. Furthermore, a quantity of type TYPE does not represent a bit sequence accessible within an MPL program, and it cannot be used as a primary in an expression.

%Derived Types%

\ blm ← -3*ilev

MPL provides several mechanisms for creating new types from existing ones. These derivative types are classified by the operations used to construct them.

Record Types

A %record type% is a composite type, values of which are called records. A record is a heterogeneous structure consisting of a fixed number of components, possibly of different types. Every record is a multiword. Components are accessed by a fixed set of distinct identifiers. Each component is called a %field%; the value of a field is a contiguous subsequence of the bit sequence that is the value of the record. Fields are individually accessible and updatable; thus a record serves as a structured collection of variables. Records are the only types that can be declared as such and thus named by an identifier in MPL(B). The declaration specifies the order of the components and the type of each. It optionally introduces an identifier, called the field %selector%, for each field. Selectors are used with records (or pointers to records) to access the bit subsequences corresponding to each field. Records can be treated as single units; alternatively, the selectors can be used to gain field-by-field access to the individual components.

In MPL(B), fields of a record described by a type declaration are packed. In the allocation of fields within such a record, words are allocated from left (relative word 0) to right, but bits within words are allocated from right (bit 35) to left. Only multiword fields are allowed to cross word boundaries, however, and those fields are aligned to minimize the number of such crossings.

The argument and result records of procedures are not packed. An entire word is allocated for each uniword quantity, and all fields are aligned to word boundaries.

The packing algorithms are subject to change in future versions of MPL.

Pointer Types

Values of a %pointer type% are called pointers. Pointers are absolute (virtual) addresses of bit sequences. A pointer value is a uniword. It is used to access a record or component of a record indirectly. The intended type of any such designated record can be specified in the declaration of a pointer variable, in which case it is called the component type or %C-type% of the pointer. Alternatively, the type can be omitted; the pointer is then said to be %C-free%. %All% pointers in MPL(B) are semantically equivalent to C-free pointers, i.e., any uniword can be assigned to any pointer variable, and any selector can be used with any pointer. No special denotations of pointer constants are predefined.

The C-type in an MPL(B) pointer declaration provides nothing more than documentation. For compatibility with future versions of MPL, however, C-free pointers should be avoided whenever possible.

On MAXC and the PDP-10, the length of a pointer value is 18

bits.

String Types

A variable of type STRING contains a pointer, the C-type of which is a system-defined record type. A record of that type is called a text descriptor and contains a set of indices plus a further pointer to a text area in which the actual characters of the string are stored. Operations involving the text descriptor or the text area are performed by a set of standard procedures, which are described in (docmps, mpsstr). String constants are written by enclosing the sequence of text characters within quotation marks. The value of a string constant is a pointer to an automatically allocated and initialized text descriptor.

Note that the basic operators of the MPL language manipulate only the pointer values, e.g., assignments can result in multiple pointers sharing the same descriptor and text area.

Array Types

An %array type% is also a composite type, values of which are called arrays. An array is a homogeneous structure consisting of a fixed number of components, all of the same type, called the component-type or %C-type% of the array. Every array is a multiword. Components are accessed by indexing. The allowable set of indices is specified in the declaration of an array by an interval constant; there is a one-to-one correspondence between integers in the interval and components of the array, which are called elements. Elements are individually accessible and updatable; thus an array serves as an indexable collection of variables, each of which has the C-type of the array.

In MPL(B), arrays are not packed. An entire word is allocated for a uniword element, and every element is aligned to a word boundary.

Array-Descriptor Types

Values of an %array-descriptor type% are uniwords, called array descriptors, which provide indirect reference to any array with a specified C-type. The C-type of an array descriptor is the same C-type. Components of an indirectly referenced array can be accessed and updated by indexing operations applied to the array descriptor. No index set is associated with an array-descriptor type; the valid indices are those that are valid for the array described by the particular array descriptor. An array descriptor contains a pointer to an array and the length of that array.

Array descriptors allow shared reference to an array; they are also useful for implementing the equivalents of "dynamic" arrays.

Procedure Types

Values of a %procedure type% are multiwords called procedure descriptors. A procedure descriptor allows indirect reference to a procedure body and also specifies the environment in which nonlocal identifiers occurring in that body are bound. Every procedure in MPL(B) accepts a (possibly empty) argument record and returns a (possibly empty) result record. Associated with every procedure type are both an argument-type or %A-type% and a result-type or %R-type%. An A-type or R-type is either empty or a record type. The procedure body referenced by a procedure descriptor should accept an argument record with fields identical in type and order to the fields of the A-type of that descriptor, and the form of its result record should similarly agree with the R-type. A procedure descriptor can be applied to an actual argument record compatible with its A-type; the indirectly referenced procedure body is thereby invoked, and the result record returned by the procedure is the resulting value.

Because of requirements for compatibility with MPL(A), MPL(B) does not distinguish among procedure types with inconsistent A- or R-types. A value with an arbitrary procedure type can be assigned to a variable with any given procedure type, and no check for correspondence between the A-type (R-type) and the argument (result) record is made when such a value is used to invoke a procedure. In MPL(B), the A- and R-types of a procedure variable thus provide nothing more than documentation.

On MAXC and the PDP-10, the length of a procedure descriptor is 72 bits.

Signal Types

Values of a %signal type% are uniwords serving as unique names for signals. These unique names are used in certain control transfers to locate and select a procedure body embedded within a special form called a catch phrase. The names are the basis for a dynamic binding mechanism. An A-type and R-type suitable for any such procedure body are associated with every signal type.

Signals are used primarily to name unusual or erroneous conditions requiring context-dependent processing.

In MPL(B), there is a single signal type, and specifications of the A- and R-type are omitted. Each can be described as a record with a single component of type WORD.

\ blm ← -2*ilev

%ATTRIBUTES AND DECLARATIONS%

\ blm ← -2*ilev

Declarations are used to specify the attributes of variables and certain constants. MPL(B) defines a heirarchy of contexts in which attribute specifications can occur. These contexts correspond to syntactic positions declaring the attributes of

array components,

record fields,

local variables of procedures and programs.

The allowable combinations of attributes are classified as component-attributes, field-attributes, and local-attributes respectively.

In MPL(B), the possible attributes of a quantity are a type and a value (either fixed or initial). With respect to allowable types, the attribute classes are properly nested and are ordered as follows:

component-attributes < field-attributes < local-attributes.

Only local-attributes can include value attributes.

This heirarchy reflects certain limitations of the MPL(B) compiler and support systems. These limitations arise because MPL(B) is an evolutionary development of MPL(A), and most will disappear in subsequent versions of MPL.

\ blm ← -3*ilev

%Constant Expressions%

constant-expr ::= expr ; -- Which has constant value

interval-constant ::=

('[_'() constant-expr '. '. constant-expr (')_']) ;

Expressions can appear within the descriptions of certain types. Any such expression must evaluate to a constant and can be constructed using only constants, predefined operators, and the standard pseudo-functions.

The value of a constant-expr is a uniword.

The endpoints of an interval constant must be non-negative integers. They can be included in or excluded from the interval by using square or round brackets respectively. For compatibility with future versions of MPL, choosing 0 as the smallest integer in the interval-constant is recommended.

Examples:

```
wordsize/charsize      -- if the sizes are defined constants
LENGTH[a] - 1          -- if the type of  a  is an array type
[0 .. 10]              -- includes 11 integers
[0 .. 10)              -- includes 10 integers
```

%Component Attributes%

```
component-attribute ::=
    "INTEGER" _
    "IN" interval-constant _
    "FIELD" _
    "POINTER" ["TO" .ID] _
    "STRING" _
    .ID ;
```

A component-attribute is used to specify the attributes of those variables that are components (i.e., elements) of an array. The uniword types INTEGER, POINTER, STRING, FIELD, and all of the set-element types are allowed. An identifier standing as a component-attribute must be a declared identifier of some record type, and any record type can be specified as a component-attribute.

The following abbreviations for commonly used set-element types are recognized:

Abbreviation	Type	
BOOLEAN	IN [0..1]	
CHARACTER	IN [0..177B]	-- ASCII characters
WORD	IN [0..777777777777B]	.

The identifier following "POINTER TO" specifies the C-type of the pointer attribute. If present, it must be a declared identifier of a record type. Note that all pointers are treated as C-free; specifying the C-type does not restrict the use of a pointer value. Such specification is allowed primarily for documentation and compatibility with future versions of MPL.

The distinction between component-attributes and field-attributes is essentially syntactic, since it is always

possible to declare a record type having a single component with any allowable field attribute.

Examples:

```
INTEGER

IN [0..wordsize)

ListNode                -- if declared as a record

POINTER TO ListNode
```

%Field Attributes%

```
field-attribute ::=

   component-attribute _

   "DESCRIPTOR" "FOR" "ARRAY" "OF" component-attribute _

   "PROCEDURE" [field-list] ["RETURNS" field-list] _

   "SIGNAL" ;

 field-list ::=

    '[ (#<',> field-attribute _ #<',>( id-list ':
   field-attribute)) '] ;

 id-list ::= #<',> .ID ;
```

A field-attribute is used to specify the attributes of a variable that is a component (i.e., a field) of a record. Any component-attribute is also a field-attribute. In addition, array-descriptor types, procedure types, and signal types are allowed.

The component-attribute following "OF" in the declaration of an array-descriptor type indicates the C-type of the elements of all arrays indirectly referenced by that descriptor.

A field-list defines the order and type of the fields within a record. It may consist of a list of field-attributes only or of a list of field attributes, each preceded by a sublist of identifiers. In the first case, the fields of the record are anonymous; in the second case, a field with the given field-attribute is introduced for each identifier in the sublist, taken in order, and the identifier serves as the selector for that field.

Anonymous and named fields cannot appear in the same field list.

Anonymous field lists are primarily useful in declaring the argument and result records of procedure variables.

A field-list immediately following the symbol "PROCEDURE" defines the A-type of the procedure type; a field-list following "RETURNS", its R-type. An omitted field-list indicates that the corresponding type is empty (not that it is unspecified; cf. pointers).

In MPL(B), any multiword of the correct length can be assigned to any procedure variable; compatibility of argument and result records is checked neither statically nor dynamically (see call-statements). For compatibility with future versions of MPL, procedure variables should be declared and used with appropriate A- and R-types.

Examples:

```
DESCRIPTOR FOR ARRAY OF INTEGER     -- indirect array access

PROCEDURE                           -- parameterless

PROCEDURE [STRING] RETURNS [INTEGER, INTEGER]

PROCEDURE [
   p: PROCEDURE RETURNS [INTEGER],
   a: DESCRIPTOR FOR ARRAY OF INTEGER,
   m, n: INTEGER]          -- two integers
```

%Local Attributes%

```
local-attribute ::=

   "INTEGER" [expr-init] _

   "POINTER" ["TO" .ID] [expr-init] _

   "IN" interval-constant [expr-init] _

   "STRING" [ '[ expr '] _ '← expr ] _

   "FIELD" [expr-init] _

   .ID _

   "DESCRIPTOR" "FOR" "ARRAY" "OF" component-attribute [ '←
expr ] _

   "ARRAY" ("OF" component-attribute array-init _
        interval-constant "OF" component-attribute) _

   "PROCEDURE" [field-list] ["RETURNS" field-list]
       [ '= procedure-body _ '← procedure-constant ] _

   "SIGNAL" [ "CODE" _ '← expr ] _

   "REGISTER" ['= constant-expr] _
```

```
"TYPE" '= "RECORD" field-list ;

expr-init ::= '← expr _ '= constant-expr ;

array-init ::= '← '[ $<',> expr '] ;

local-declaration ::= id-list ': local-attribute ;
```

A local-declaration introduces one or more identifiers and defines a set of attributes for each. Local declarations appear at the head of a program, data, or procedure module (see MODULES). An identifier can be equated by declaration to some constant value, in which case the identifier denotes that constant value and cannot subsequently be updated. Otherwise, each identifier introduces a new variable, the type and initial value of which are determined by the local-attribute used in its declaration.

Any component-attribute or field-attribute is an acceptable local-attribute, in which case the declaration introduces an uninitialized local variable. In addition, a local-attribute can specify the value to which a symbolic constant is to be equated or with which a variable is to be initialized. When the initialization is given by an expr-init, the expression must evaluate to a uniword, and the default conversion rules apply. The following more specialized initializations of uniwords are available:

The attribute STRING followed by a bracketed expression introduces a string variable and specifies the automatic creation of a text descriptor, a pointer to which is the initial value of that variable. In addition, an empty text area large enough to contain the number of characters given by the value of the bracketed expression is allocated. No text area is allocated if the value of the expression is zero.

The lifetimes of these storage areas are identical to the lifetime of the string variable, and they are automatically deallocated.

An attribute SIGNAL CODE generates a unique name of a signal, which serves as the initial value attribute. A variable with such an attribute cannot subsequently be updated.

An identifier standing as a local-attribute must be the identifier of a declared record type and introduces a multiword variable with that record type. No associated initialization can be specified.

The attribute following "OF" in a local-attribute for an array type specifies the C-type of the array. The interval serving as the index set can appear explicitly, following "ARRAY", in which case no initialization can be specified. Alternatively, a list of N initial values can appear, in which case the index set is [0..N).

Each expression in the initialization list must evaluate to a bit string which is compatible with the C-mode of the array according to the default conversion rules.

The attribute PROCEDURE can be used to equate an identifier to a procedure body. Within that procedure body, the fields of the argument record are accessed by the identifiers introduced as selectors in the field-list defining the A-type of the procedure type (see MODULES).

The attribute REGISTER is used to associate an identifier with an absolute address in memory. Registers are used as variables of type WORD. They are shared across all contexts. If a constant-expr appears, its value is taken as the absolute address of the register. Otherwise, an arbitrary MAXC/PDP-10 accumulator is associated with the identifier.

Such registers are removed from the pool used for expression evaluation within the scope of the REGISTER declaration but are %not% automatically saved and restored across transfers between contexts.

The TYPE attribute can only be used to name and describe fixed record types. The field-list specifies the type, relative position, and (optionally) the selector of each field as described under field-attributes.

Association of identifier lists and local attributes is constrained by the following rules:

An identifier can be declared only once in each module (see MODULES).

An identifier equated to a procedure body can be declared only in a program module (i.e., procedure declarations do not nest), and only one identifier can appear in the id-list.

Only one identifier can appear in the id-list equating an identifier to a record type.

A declaration of a SIGNAL CODE cannot appear in a procedure module.

Examples:

```
n: INTEGER                          -- a single integer

i, j, k: INTEGER ← 0                -- i ← 0; j ← 0; k ← 0

nc: INTEGER = wordsize/charsize

r: STRING                           -- a pointer variable only

s: STRING[0]                        -- descriptor
```

```
        t: STRING[100]                  -- descriptor, 100 char text
        area

        table: DESCRIPTOR FOR ARRAY OF INTEGER

        a: ARRAY OF INTEGER ← [2,3,5,7]  -- a[0], ..., a[3]

        b: ARRAY [0..nc) OF CHARACTER    -- unpacked "word" of
        characters

        f: PROCEDURE [n: INTEGER] RETURNS [INTEGER] =
           BEGIN  -- an MPL factorial function of course --
           x: INTEGER ← 1;  i: INTEGER;
           FOR i THRU (1..n] DO  x ← x*i;
           RETURN[x]
           END

        g, h: PROCEDURE RETURNS [success: BOOLEAN]

        ListNode: TYPE = RECORD          -- a two component record
          [NodeValue: INTEGER,
           NextNode: POINTER TO ListNode]

        root: POINTER TO ListNode

        node: ListNode                   -- a local record

        DataOverflow: SIGNAL CODE        -- a signal definition

        mplUnwind, STRBoundsFault: SIGNAL   -- bindable signal
        variables

        r1, r2: REGISTER                 -- distinct unspecified
        accumulators

        page1: REGISTER = 1000B          -- the first word of page 1
```

\ blm ← -2*ilev

%MODULES%

A module is the collection of declarations and statements with which a set of local variables is associated. An instance of such a module, with a specific set of local variables and some control information, is called a %context%. Associated with every context are a code body and a %frame%. A frame is an implicitly declared record containing some hidden fields used for control information as well as a field for each declared variable. A (possibly null) %argument record% is supplied when a context is created and remains associated with that context throughout its lifetime.

\ blm ← -3*ilev

%Module Classification%

```
body ::= $(locals ';) $(block-dcl ';) $<';>statement ;

program-module ::=

   .ID ': "PROGRAM" [field-list] '= prog-body '. ;

 prog-body ::= "BEGIN" [include-list ';] body "END" ;

data-module ::=

   .ID ': "DATA" [field-list] '= data-body '. ;

 data-body ::= "BEGIN" $(locals ';) "END" ;

procedure-body ::= "BEGIN" body "END" ;

locals ::= local-declaration _ base-declaration ;
```

A procedure-module is not defined as a syntactic unit but consists of a local-attribute equating an identifer to a procedure-body. The identifier is the name of that module.

Three kinds of modules are distinguished in MPL(B): program modules, data modules, and procedure modules. These three classes of modules give rise to corresponding classes of contexts. Program and data modules are the units of compilation and are terminated by a ".". Program-contexts are explicitly created from program modules and explicitly destroyed using a set of procedures provided by the MPS support package (see ?). Their lifetimes are arbitrary. Data-contexts are similar to program-contexts, but they have null code bodies. They are intended to serve as carriers of data shared by several program-contexts. Procedure modules are declared within program modules and thus can share a set of nonlocal variables, namely the locals of a program-context. Creation of a procedure-context and transfer to the procedure body are inseparable actions, as are return and destruction of the procedure-context's frame. Thus there is a nested set of procedure-contexts for each program-context, and the lifetimes of procedure frames follow a LIFO discipline.

Any module body can be headed by local declarations which introduce and define identifiers as described in the previous section. The scope of each identifier is the text of the module following the identifier list in which it appears. In the case of a program module, however, the scope of an identifier excludes any scope defined by an embedded procedure module in which the same identifier is redeclared, and such redeclaration is not considered to introduce a conflict.

In particular, the scope of an identifier includes any initializing expression; thus definitions are recursive. An identifier cannot, however, be used before it is declared unless its type is a procedure type.

Examples:

```
a: INTEGER ← a+1        -- nonsense, but legal

b: TYPE = RECORD[b1: INTEGER, b2: POINTER TO b]     -- ok

c: TYPE = RECORD[c1: INTEGER, c2: c]        -- not
allowed
```

The fields of the argument record associated with a module are introduced and defined by the field-list immediately following the symbol "PROGRAM", "DATA", or "PROCEDURE". Within the body of the module, the fields of that record are accessible as local variables, i.e., the identifier of each field itself designates the corresponding record component, and the argument record is not named explicitly. These identifiers are considered to be local to the module and are governed by the same scope rules as any identifiers introduced by local declarations in the same module.

In MPL(B), the result records of procedure modules are treated similarly, and any explicitly named fields of such records can be used as local variables. For compatibility with future versions of MPL, such use should be avoided.

Note that the scope of an identifier appearing in a field list defining the A-type or R-type of a procedure body is the corresponding procedure module, not the program module. The scope of the identifiers of fields of an explicitly declared record type, however, is the same as that of the identifier of the record type itself.

A data-body consists only of declarations. The variables declared within a data module can be made known within a program module (and its embedded procedures) by an include-list at the head of the program module (see Included Modules).

The bodies of program and procedure modules additionally contain a list of statements. Transfer of control to such a body causes those statements to be executed. In the absence of explicit transfer- or jump-statements, the statements in the body are executed in the order in which they appear.

The head of a module can also specify the processing of any signal generated or propogated as a result of its execution (see Signals and Blocks).

%Included Modules%

```
include-list ::= "DIRECTORY" #<',> (.ID ': "INCLUDE" link) ;

link ::= '( .ID ', .ID ', ') ;

base-declaration ::= "BASE" #<',> base-id ;

base-id ::= .ID "ON" .ID ;
```

An include-list appearing at the head of a program module makes the definitions of identifiers declared in specified data modules available within that program module. The links in the include-list name files containing the (compiled) text of the data modules. Any local redefinition of a particular identifier makes the included definition inaccessible.

A link with the form (directory, file,) references the highest-numbered version of the TENEX file <directory>file.MPS .

All actual accesses to variables (but not to constant definitions) in an included module must specify a pointer to the frame of a particular data-context. That pointer can be specified explicitly as part of the reference (see VARIABLES); alternatively, the appropriate pointer to be used over some sequence of statements can be established by a base declaration, and the variables within the data-context can be named directly within that sequence. In the base-id, the first identifier specifies a module name introduced in the include-list, and the second identifier must designate a simple variable. Within the range of the base declaration, the specified variable should contain an appropriate frame pointer, and it is reevaluated as part of every access.

No special significance is given to a module name introduced in an include-list; it need bear no relation to either the identifier at the beginning of the referenced data module or the file name.

Examples:

```
-- include list

   DIRECTORY
      Registers: INCLUDE (mps, mplregs, ),
      DataArea: INCLUDE (directory, shareddata, )

-- base declaration

   BASE DataArea ON ptr
      -- if v is a variable local to DataArea, reference to  v
   is then equivalent to  ptr./v --
```

```
\ blm ← -2*ilev
```

%VARIABLE FORMS%

A variable is an updatable storage area. There is a well-defined type associated with every variable. The type determines the length of any bit sequence stored in that variable and implies an interpretation of that bit string.

```
var ::=

   .ID _ array-component _ record-component ;

array-component ::= var '[ expr '] ;

record-component ::=

   var qualifier _

   '( expr ') '. field ;

 qualifier ::= ('$ _ '.) field ;

 field ::= field-constant _ .ID ;
```

An identifier of a declared variable denotes that variable within the scope of the identifier, and the type of the variable is that type specified in its declaration.

An identifier declared within an included module can stand alone as a variable only if it is a register or is in the range of a base-declaration for that module.

Indexing can be used to select a component of an array. The bracketed index expression must evaluate to a uniword. The type of the variable must be either an array type, in which case a component of that variable is selected directly, or an array descriptor type, in which case a component of the array indirectly referenced by the descriptor is selected. In either case, the type of the component variable is the C-type of the array or array descriptor.

The value of the index expression should fall within the declared index set of the array, but MPL(B) provides no automatic bounds checks.

A qualifier can be used to select a component of a record. The record can be specified by a variable with a record type, in which case the operator "$" is used to select a component, or by the value of a pointer to a record, in which case the operator "." is used to select a component. If the pointer is not itself the value of a variable, the expression defining its value must be parenthesized. The field constants used in selectors have the following forms (see also Constants):

An identifier of an explicitly declared record type. Such a field selects an initial bit sequence with length equal to the length associated with the record type, and the type of the resulting variable is that record type.

Use of a record type as a selector allows the bit string representing an entire record to be referenced through a pointer.

An identifier of a field of an explicitly declared record type. The bit sequence with length and position corresponding to the

named field is selected from the variable (which should have a length equal to the length associated with the record type). The type of the selected variable is the declared type of the field.

The form / .ID , which identifies a field of an implicitly declared record (i.e., a frame) and has the same interpretation as the identifier of an explicitly declared field.

If f is a pointer to the frame of an included data module with local variable x , the form f./x refers to that variable (which is more simply named by the form x within the range of a base-declaration for that module).

A constructed field constant, in which case the type of the resulting variable is an appropriate set-element type.

In addition, the value of an identifier declared to be a field variable can be used to select a component variable, the type of which is an unspecified uniword type.

Field selection can be iterated, but all qualification must be explicit. If r is a record containing a field f1 , the type of which is a record type with field f2 , that field must be referenced by the form r$f1$f2 , not r$f2 .

In MPL(B), there is no check that the variable to which a selection operation is applied has a record type or that the qualifier is appropriate for the variable.

Examples:

i	-- an integer variable
a	-- an array variable
a[i]	-- an integer variable
node	-- a record variable
node$NextNode	-- a pointer variable
ptr.NextNode	-- ditto
(hash[w]).ListNode	-- a record variable

%EXPRESSION FORMS%

\ blm ← -3*ilev

%Expressions%

expr ::= conditional-expr _ disjunct ;

The value of an expression is a sequence of bits. Unless the

expression is a variable or a function call, that bit sequence is a uniword. Default conversion rules apply to all uniword operands of the operators discussed in this section.

%Selection Operators%

```
conditional-expr ::= if-expr _ case-expr ;
```

A conditional expression allows the value of one expression to be used to select another expression from a list of alternatives.

If-Expressions

```
if-expr ::= "IF" expr "THEN" expr "ELSE" expr ;
```

In an if-expr, the expression following "IF" is evaluated. If that expression is true (has nonzero value), the expression following "THEN" is evaluated, and its value is taken as the value of the conditional expression. Otherwise, the expression following "ELSE" is evaluated. All expressions must evaluate to uniwords.

Examples:

```
IF n > 1 THEN sum/(n-1) ELSE 0

IF x < 0 THEN -1 ELSE IF x > 0 THEN 1 ELSE 0
```

Case-Expressions

```
case-expr ::=

   "CASE" sum

      ("OF" $case-of-expr _ "IN" $case-in-expr _ relational
   $case-op-expr)

      "ENDCASE" expr ;

 case-of-expr ::= #<',>(binrel _ sum) ': expr '; ;

 case-in-expr ::= #<',>(interval _ char-class) ': expr '; ;

 case-op-expr ::= #<',>sum ': expr '; ;
```

In a case-expr, the expression following the "CASE" is evaluated, and that value becomes the left operand of the relations labeling the various alternatives. The expression following the first relation which is satisfied is evaluated, and its value is taken to be the value of the conditional expression. All other expressions are skipped, even if relations labeling subsequent alternatives are satisfied. If none of the relations is satisfied, the expression following "ENDCASE" is selected.

In a case-of-expr, each label specifies a list of items,

and each item consists of a relational operator (or an implicit "=" if only a sum appears) and a right operand. The labeled expression is selected if any of the relations is satisfied.

In a case-in-expr, each label specifies a list of intervals. The labeled expression is selected if the left operand falls within any of the intervals.

In a case-op-expr, the relational operator is supplied with the left operand, and each label specifies a list of right operands.

All expressions must evaluate to uniwords.

Examples:

```
CASE i OF
   a, b, c: j+k;        -- i=a OR i=b OR i=c
   >(n+1), <0: j+1;     -- i>n+1 OR i<0
   =j, #k:  f[i+1];     -- i=j OR i#k
   =j: i/0              -- never evaluated (see above)
   ENDCASE 1

CASE char IN    -- character translation
   [0..37B]: ' ';            -- control characters to blank
   LL:  char + ('A-'a);      -- lower case to upper case
   ENDCASE char

CASE x <    -- a "step" function
    0:  0;
   10:  1;
   25:  2;
   50:  3;
   ENDCASE 4
```

%Logical Operators%

In MPL, the Boolean values TRUE and FALSE are represented by the integers 1 and 0 respectively. Any uniword is acceptable in a context requiring a truth-value, however, and the expression is considered to be true if, and only if, its value has at least one nonzero bit.

```
disjunct ::= #<"OR"> conjunct;

conjunct ::= #<"AND"> negation;

negation ::= ["NOT" _ '¬] relation;
```

A disjunct is true if, and only if, some conjunct is. The value of a disjunct is defined as follows: the form p OR q is equivalent (except syntactically) to the form

IF p THEN TRUE ELSE q .

Thus the conjuncts are evaluated from left to right, and evaluation terminates with the first true conjunct.

A conjunct is true if, and only if, all negations are. The value of a conjunct is defined as follows: the form p AND q is equivalent (except syntactically) to the form

IF p THEN q ELSE FALSE .

Thus the negations are evaluated from left to right, and evaluation terminates with the first false negation.

The operator "¤" (or "NOT") inverts the Boolean value of its operand.

```
relation ::= sum [binrel];

 binrel ::=

   ["NOT" _ '¤] ( relational  sum _ "IN" (interval _
   char-class));

 relational ::= '> '= _ '< '= _ '> _ '< _ '= _ '#;

 interval ::= ('( _ '[) expr '. '. expr ('] _ '));

 char-class ::=

   "CH"  _  -- any character

   "ULD" _  -- upper case letter or digit

   "LLD" _  -- lower case letter or digit

   "LD"  _  -- lower or upper case letter or digit

   "UL"  _  -- upper case letter

   "LL"  _  -- lower case letter

   "L"   _  -- lower or upper case letter

   "D"   _  -- digit

   "PT"  _  -- printing character

   "NP" ;   -- nonprinting character
```

The relational operators stand for the following relations:

<	less than	<=	less than or equal
>	greater than	>=	greater than or equal
=	equal	#	not equal

They are defined by the usual ordering obtained by interpreting their operands, which must be uniwords, as (signed) integers.

Note that the relational operators should not be used for strings, since they then specify the comparison of pointers, not character sequences.

An interval is the set of all integers in some range. The endpoints of the interval are specified by a pair of expressions, which must evaluate to uniwords. An endpoint may be included (indicated by a square bracket) or excluded (indicated by a round bracket). The operator "IN" tests for membership in the specified interval, e.g.,

f[x] IN [m..n)

is true if, and only if, m <= f[x] < n . It differs from

(f[x] >= m) AND (f[x] < n)

in that the function f is called only once.

The char-classes provide convenient notation for intervals or sets of intervals corresponding to the indicated classes of ASCII characters.

Examples:

```
x # y

x ¤= y     -- equivalent to x # y

NOT x = y     -- ditto

char IN LLD     -- tests for lower case letter or digit

(char IN ['a..'z]) OR (char IN ['0..'9])  -- as above  (in
ASCII)

char >= 'a AND char <= 'z OR char >= '0 AND char <= '9  --
ditto
```

%Arithmetic Operators%

The operands of arithmetic operators are interpreted as signed binary numbers and, within the limitations of the representation, the results are the signed binary numbers given by the usual rules of arithmetic.

The operands of the bit string operators are interpreted as sequences of bits, extended to the machine's word length, and the results are obtained by bit-wise combination of corresponding elements of the sequences.

In either case, a Boolean expression is a suitable operand; TRUE

and FALSE have the integer values 1 and 0 respectively.

```
sum ::= #< '+ _ '- > prod;

prod ::= #< '* _ '/ _ "MOD" > bitor;
```

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

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

The form i MOD j gives the remainder of i/j .

```
bitor ::= #<"BITOR" _ "BITXOR"> bitand;

bitand ::= #<"BITAND"> factor;
```

The operators "BITOR", "BITXOR", and "BITAND" specify the bit-by-bit logical-or, exclusive-or, and logical-and of their operands, respectively.

```
factor ::= ['-] prim;
```

The unary operator "-" negates its argument.

Examples:

```
-i + j*k MOD 2 - 3    -- parsed as ((-i)+((j*k) MOD 2))-3

i * j BITAND 777B BITOR 2B    -- i*((j BITAND 777B) BITOR 2B)
```

%Primaries%

primaries are the syntactic entities which can appear as operands without further parenthesization. The value of any primary is a bit string, the length of which depends upon the form of the primary. Certain primaries are the only expression forms which can yield multiword values.

```
prim ::=

    var _

    constant _

    function-call $qualifier _

    assignment-expr _

    '@ var _

    pseudo-function _

    rhs-statement _
```

```
catchp-rhs _

'( expr ') $qualifier ;
```

The value of a variable standing as a primary is the bit sequence which is the current value of that variable. The length of the bit sequence is equal to the length associated with the type of the variable.

The value of the form @ v is a pointer which is the address of the variable v . The unary operator "@" is undefined if the variable v is not represented by an integral number of machine words.

In the case of a component variable, the rightmost qualifier must be a constant field which selects an integral number of words.

Parentheses can be used to control the association of operators and operands within expressions. Parenthesization of an expression does not change the length of the bit string which is the value of that expression. Note that it is possible to extract a field from the value of a parenthesized expression or from the value returned by a function.

Examples:

```
@i     @a[i]      @ p.HeaderNode$linkword

((x+y)*z)

(base+disp)$righthalf    -- addition mod FIELDMAX[righthalf]+1
```

Special Primaries

```
pseudo-function ::=

    "FIELDMAX" '[ .ID '] _

    "RECORDSIZE" '[ .ID '] _

    "DISPLACEMENT" '[ .ID '] _

    "DESCRIPTOR" '[ .ID '] _

    "LENGTH" '[ var '] _

    "DIV" '[ prod ': var '] _

    ("MIN" _ "MAX") '[ #<',>expr '] ;
```

Pseudo-functions have the syntactic forms of function calls, but their meanings are predefined.

The form FIELDMAX[f] has a value equal to the largest integer which can be stored in the field f . The

identifier must designate a field constant, and the number of bits in that field cannot exceed the word length of the machine.

The form RECORDSIZE*[r]* has a value equal to the number of machine words used to represent a variable of type r . The identifier must designate a type which is a record mode.

The form DISPLACEMENT*[f]* has a value equal to the relative word address of the field f . The identifier must designate a field constant.

The form DESCRIPTOR*[a]* has a value which is an array descriptor for the array a . The identifier must designate an array variable.

The form LENGTH*[v]* has a value which is equal to the number of array elements in the variable designated by v . The type of the variable must be either an array type or an array-descriptor type. In the latter case, the value is equal to the length of the array indirectly referenced by the descriptor that is the value of the variable.

The pseudo-function "DIV" requires as its argument an expression in which the principal connective is the operator "/". The value of the pseudo-function is the quotient of the division; the remainder is assigned to the variable which follows the ":".

The pseudo-functions "MIN" and "MAX" select the algebraically smallest and largest values respectively from a list of expressions. Operands are interpreted as signed binary numbers. These operators are not defined for multiword operands.

rhs-statement ::= signal _ system-call ;

These two statement forms (q.v.) produce values which can be used within expressions.

catchp-rhs ::= "CODE" _ "MESSAGE" ;

Within a catch phrase, the values of the primaries "CODE" and "MESSAGE" are respectively the internal name of the signal being processed and the message associated with that signal. Their values have the attributes associated with types SIGNAL and WORD respectively.

%Constants%

A constant denotes a fixed string of bits. The length of the bit string and the mapping from the denotation of the constant to the value of the bit string are determined by the syntactic form of the constant.

```
constant ::=

    literal _

    string-constant _

    procedure-constant _

    field-constant _

    .ID ;
```

An identifier equated by declaration to some constant-expr is itself a constant, and its value cannot be updated. Such a constant is a uniword, and its value is equal to the value of the constant-expr.

Literals

```
literal ::= .NUM _ "TRUE" _ "FALSE" _ char ;

 char ::=

   .SR1  _

   "SP"  _     -- a space

   "EOL" _     --  TENEX CR-LF (37B)

   "ALT" _     -- a TENEX ESC

   "CR"  _     -- a carriage return

   "LF"  _     -- a line feed

   "FF"  _     -- a form feed

   "TAB" _     -- a tab

   "EOS" ;     -- an end-of-string, see (docmps,mpsstr)
```

The value of a literal is a bit string which is a uniword.

Numeric literals are written as ordinary decimal or octal numbers. Such a literal denotes an integer value with the same length as an INTEGER variable.

> On MAXC and the PDP-10, a decimal literal cannot exceed 34359738367 in magnitude, and an octal literal cannot contain more than 12 digits (excluding leading zeros).

The BOOLEAN constants "TRUE" and "FALSE" have the values 1B and 0B respectively and the same length as a BOOLEAN variable.

The value of a character literal is the positive integer which is the ASCII encoding of the character quoted in or denoted by

that literal. The length of the literal is equal to the length of a CHARACTER variable.

Examples (all examples on each line have the same value):

1000 1000D 1D3 1750B 175B1

SP ' 40B

String Constants

string-constant ::= .SR ;

The value of a string constant is a pointer to a fixed text descriptor which describes a text area containing the sequence of characters in the quoted string.

The text descriptor and the text itself may be shared by other string constants. These auxiliary data structures are not updatable. Their lifetimes are identical to the lifetime of the program-context in which the string constant appears.

Examples:

"a string constant"

"" -- the null string

"""" -- a string containing a single quote character

Procedure Constants

procedure-constant ::= "PROCEDURE" .ID ;

The value of a procedure-constant is a multiword which serves as a procedure descriptor. The descriptor contains two components, which encode the following information:

The entry point of the code for the procedure body.

The program-context in which nonlocal identifiers occurring within the procedure body are to be bound.

If the procedure-constant appears in a scope in which the identifier is equated by declaration to a procedure body, the value is a descriptor of that body bound to the program-context in which the procedure-constant is embedded. Otherwise, the identifier is an external procedure identifier, and the descriptor is considered to be "unbound" until it is completed dynamically by a run-time binding facility (see call-statements).

Field Constants

field-constant ::=

.ID _

'/ .ID _

"FIELD" '[[constant-expr ',] constant-expr ':
constant-expr '] ;

The value of a field constant is a bit pattern which can be used to select a substring of bits from a bit string. The value itself is a uniword.

On MAXC and the PDP-10, values of type FIELD are represented by byte pointers.

When a field constant is used as a primary, the length of the substring to be selected by that constant cannot exceed the word length. Direct selection of a multiword component using a field constant as a qualifier is, however, possible (see VARIABLE FORMS).

An identifier standing as a field constant must name either an explicitly declared record type or a field of such a record type. In the first case, the value is a selector which extracts an initial bit string with length equal to the length associated with the record type. In the second case, the value is the selector which extracts the bit substring corresponding to the named field from a bit string with length equal to the length of the record type?

The unary operator "/" constructs a similar selector for a named field of an implicitly declared record, such as a data segment described by an INCLUDE module or a procedure frame.

The final form of a field constant allows the selector to be specified explicitly. The form FIELD [w, b : t] has a value which selects the b bits beginning L*(w+1) + (b+t) bits from the left end of the bit string, which should have a length of at least L*(w+1) bits. L is the word length of the machine.

If w is omitted, a value of 0 is assumed.

The values of b and t must be nonnegative with sum not exceeding L .

For MAXC and the PDP-10, the value of L is 36.

Examples:

NextNode -- if declared as a field

/i -- if declared as a variable

FIELD[0,18:18] -- the left half of a PDP-10 word

%Function Calls%

function-call ::= (.ID _ record-component) args ;

A function call specifies that a procedure is to be invoked with given arguments; it can also specify the action to be taken for any signal generated or propogated by the called procedure. Construction of the argument list, transfer of control, and processing of signals in function-calls and call-statements (q.v.) are identical.

The called procedure returns a result record constructed by execution of a return-statement. In MPL(B), the value of the function-call is determined as follows:

If the function-call is qualified, the field of the result record selected by the qualifier is extracted, and the attributes of the extracted value are determined by the type of the extractor.

If the function-call appears without qualification as the right-hand side of an assignment statement, an initial field or set of fields are extracted from the result record and assigned as determined by the left-hand side of the assignment statement (q.v.).

In all other cases, the value of the function call is that uniword which is the first word of the result record, and the remainder of the result record, if any, is discarded.

Note that the R-type of the procedure is ignored in the application of these rules.

In particular, the value of a function-call intended to return a multiword will be interpreted incorrectly if it appears without qualification in an argument list.

Example:

If x is a uniword and q is a field with a procedure type,

f[x, p[x]]

is a call of f with an argument record constructed from two uniwords, but

f[x, p[x]$q]

is a call of f with a uniword and a procedure descriptor.

For compatibility with future versions of MPL, uses of function-calls as primaries should be restricted to the first two cases unless the result record constructed by the called procedure has a single component which is a uniword.

Examples:

ReadCharacter[]

gcd[m, n]

CompareString[s, token[i]]

%Assignments%

An assignment specifies that an expression is to be evaluated and the result assigned to a variable. An assignment itself has a value and can be used within an expression.

assignment-expr ::= var '← expr _ var '← ': expr ;

The value of an assignment form using the operator "←" is the value of the expression; as a side effect, that value is also assigned to the variable. An assignment using that operator yields a value and can be used as a primary only if the assigned value is a uniword. The default conversion rules apply.

The value of an assignment form using the operator "←:" is the value of the variable prior to its updating. As a side effect, the expression on the right is evaluated and assigned to the variable as described above. The assignment operator "←:" is defined only for uniwords, and default conversion rules apply.

If an unqualified function-call stands as the right-hand expression, the first word of the result record returned by the function is assumed to be the value of the function.

For compatibility with future versions of MPL, this form should not be used when the function in fact returns more than a single uniword in the result record (see function-calls), but qualification can be used to select a particular uniword.

Note that syntactic ambiguities involving the assignment operator are resolved by taking the longest possible expression as the right operand.

Examples:

```
a ← b ←: a              -- exchange a and b (value: old b)

table[n←n+1] ← 0        -- table[n] = 0

table[n←:n+1] ← 0       -- table[n-1] = 0

i ← j + k ← 3           -- formally ambiguous

i ← j + (k ← 3)         -- equivalent to the above
```

Note: (i ← j) + (k ← 3) is a sum and quite different.

\ blm ← -2*ilev

%STATEMENT FORMS%

\ blm ← -3*ilev

%Statements%

statement ::= $label (simple-statement _ compound-statement) ;

simple-statement ::=

assignment-statement _

transfer-statement _

jump-statement _

bump-statement _

conditional-statement _

do-statement _

misc-statement _

null ;

A parenthesized identifier standing before a statement serves to label that statement, and the identifier is thereby declared to be a label. Jump statements embedded within such a labeled statement can be used to specify repetition or termination of that statement. A label can also serve as an operand of an inline instruction anywhere within the scope of that label.

Examples:

```
(init) (reset) n ← 0      -- the labels are just comments

(loop) DO
   BEGIN   char ← ReadCharacter[];
   IF char = EOL THEN  EXIT loop;
   AppendChar[line, char]
   END

(insert) MakeEntry[table, name
  !DuplicateName:
      EXIT insert;
   TableOverflow:
      BEGIN  Expand[table]; REPEAT insert
      END ]
```

%Assignment Statements%

assignment-statement ::=

```
var '← expr _

extractor '← (var _ function-call) ;

extractor ::= '[ #<',>var '] ;
```

An assignment statement specifies that a right-hand side is to be evaluated and the result assigned to a left-hand side which is either a variable or a bracketed list of variables, called an extractor. In the latter case, disjoint fields of a single computed bit sequence are assigned to the various variables.

An assignment operation with a single variable as the left-hand side is defined for bit strings of any length, and the default conversion rules apply.

If an unqualified function-call stands as the right-hand expression, the number of words implied by the length of the variable on the left is copied from the result record, i.e., the function is assumed to return a record with a single component, the type of which has the same length as the type of the variable.

For compatibility with future versions of MPL, this form of assignment should not be used unless the function does return such a record. MPL(B) does not, however, even check that all the copied words are part of the result record.

When the left-hand side is an extractor, the right-hand side must be either a variable with a record type or a function call.

In the case of a record, successive fields are extracted from the record value and assigned to variables in the extractor. Processing of both the record fields and the extractor proceeds from left to right, and the number of variables in the extractor cannot exceed the number of fields in the record. All unused fields of the record are ignored. The assignment is illegal if any of the subassignments are illegal according to the default conversion rules.

In the case of a function value, processing is similar, but the selection of fields from the bit string returned by the function is controlled by the types of the variables in the extractor. The selected fields are in exact correspondence with the fields of the return record if the values of the latter fields, taken in order, could validly be assigned to the variables in the extractor without violation of the default conversion rules. In other words, identity of the individual components of the result record is properly maintained if a record with the R-type of the procedure could validly be assigned to the same extractor.

For compatibility with future versions of MPL, this correspondence rule should be observed. MPL(B) in fact takes a single word from the return record for every uniword variable in the extractor and exactly the required

number of words for every multiword in the extractor.

Examples:

```
char ← '*

table[i] ← v

p1 ← PROCEDURE Scan  -- a multiword assignment

[v, ptr] ← node     -- v ← node$NodeValue; ptr ← node$NextNode

[time, date] ← ReadClock[]
```

%Transfer Statements%

```
transfer-statement ::=

   call-statement _

   return-statement _

   stop-statement _

   signal-statement ;
```

A transfer statement specifies transfer of control from one context to another and can also cause a context to be created or destroyed.

Call Statements

```
call-statement ::=

   (.ID _ record-component) args _

   "CALL" (.ID _ record-component) ;
```

A call-statement specifies the invocation of a procedure. The identity of the procedure is determined by the identifier or record-component. The argument list is optional, but if it is omitted, the "CALL" must appear. After any required argument record has been constructed, a new procedure context is created and receives control. A return-statement (q.v.) executed by that procedure destroys the new context and returns control to the point of call, possibly supplying a result record to the caller. The argument list can additionally specify how any signals generated or propogated by the called procedure are to be processed. When a procedure is invoked by a call-statement, any result record is discarded.

The type of the identifier or record component must be a procedure type. The identity of the procedure to be invoked is determined as follows:

If the call-statement appears in a scope in which the identifier is equated by declaration to an explicit procedure body, that body is invoked. Its globals are bound to the variables declared in the program-context in which the call-statement is embedded.

If the identifier is bound to a procedure variable or if a record component is specified, the procedure body described by the procedure descriptor which is the current value of that variable is invoked. Note that the descriptor selects not only the procedure body but also the program-context in which the globals occurring in that body are bound.

If the identifier is not otherwise declared, it is assumed to designate a procedure body defined externally to the program module in which the call occurs. The identity of that %external procedure% depends upon the environment in which the call occurs; it is determined dynamically prior to or as part of the first call of that external procedure.

An external procedure is implemented as an implicitly declared procedure variable, the scope of which is the entire program.

There are standard facilities within MPS for binding some or all external procedures "by name", i.e., by matching the external procedure identifier to an identical identifier equated in some other program-context to a procedure body (see ?).

args ::= '[$<',>expr [':' catch-phrase] '] ;

Arguments are transmitted to the called procedure as components of an argument record constructed by the caller. The field-list defining the A-type of the procedure body specifies the format of the argument record with respect to the called context, and the identifiers introduced by that field-list thus select fields of the record supplied by the caller (see MODULES).

The argument record is constructed by evaluating the expressions in the argument list and concatenating the resulting bit sequences. The identifier of the i-th field in the record defining the A-type of the called procedure selects the value of the i-th expression used to construct the argument record if the expressions, taken in order, could validly be assigned to successive fields of a record with that A-type. In other words, the identity of the individual components of the argument record is properly maintanined if the expressions evaluate to bit strings compatible with corresponding fields of the record according to the default conversion rules.

For compatibility with future versions of MPL, this correspondence should be maintained. The exact rule for argument record construction in MPL(B) is that the

expressions are evaluated from left to right, and the length of any uniword is extended to the word length by prefixing zero bits.

Note that the A-type of the procedure to be called is ignored in the construction of argument records.

Catch phrases are discussed in connection with signals.

Examples:

```
CALL ScanID

TypeCharacter[char]

AppendString[s, t
  !STRBoundsFault:
      BEGIN  ExpandString[s]; RESUME
      END ]
```

Return Statements

```
return ::= "RETURN" ['[ #<',>expr ']];
```

A return statement specifies the deletion of a procedure-context and a return of control to the calling context. It can also specify the construction of a result record to be returned to the caller. A return-statement can appear only within a procedure body.

The return record is constructed by evaluating the expressions in order and concatenating the resulting bit sequences. If no expression list appears, the result record is empty.

The rules of construction are identical to the rules used for constructing argument records. For compatibility with future versions of MPL, the number of expressions should be identical to the number of fields in the record type which is the R-type of the procedure body, and the lengths of corresponding expressions and fields should be compatible.

Extraction of fields from the result record by the caller is discussed in connection with function-calls and assignments.

Return from a procedure-context releases the frame and any other automatically allocated storage for that context. Other local storage becomes inaccessible and should be released prior to return. Any pointers to automatically allocated storage become meaningless.

Examples:

```
RETURN

RETURN[TRUE, table[i]]
```

Stop Statements

```
stop ::= "STOP" ;
```

A stop-statement causes control to be transferred from the program-context in which the stop-statement is executed to the program-context which is its parent. The frame of that program-context is not deallocated. If control reenters the stopped context, execution continues with the successor of the stop-statement. A stop-statement cannot appear within a procedure body.

Signal Statements

```
signal-statement ::= signal-call _ resume ;
```

Signal statements provide a mechanism for transferring control in which the target context is determined dynamically by matching a unique name. The code that is executed as a result of such a transfer is called a catch phrase.

```
signal ::=

   ("SIGNAL" _ "ERROR") '[ sig-list '] ;

 sig-list ::= expr [', expr] ['! catch-phrase] ;

catch-phrase ::= #<';> catch-case [';  catch-all] _
catch-all ;

 catch-case ::= #<',> .ID ': statement ;

 catch-all ::= "ANY" ': statement ;
```

A signal-call specifies that a signal is to be generated. The value of the first expression in the sig-list must be a uniword and is taken as a unique name. Signal generation causes the successive activation of catch-phrases. propogation of a signal begins in the signaling context and continues from a procedure-context to its calling context or from a program-context to its parent context. In each context, the catch phrase appearing in the argument list of the call by which control left the context (except the signaling context) is inspected first, followed in reverse order by the catch phrases at the heads of all blocks textually enclosing that call (or the signal-call in the case of the signaling context). The first catch-case labeled by an identifier evaluating to the unique name or the first catch-all encountered in each context is executed. Signal propogation continues until one of the following actions occurs:

A resume statement (q.v.) is executed, in which case propogation of the signal is terminated and control returns to the signaling context. The resume statement can specify

a value, which is the value of the signal-call.

A jump-statement or return-statement within a catch-phrase and leading out of that catch-phrase is encountered. Prior to execution of such a statement, a special "unwind" signal is generated on behalf of the signaling context and is propogated back to the context containing the jump- or return-statement, which is then executed. Each context which allows the unwind signal to propogate is deleted.

If the signal-call specifies "ERROR", the continuation implied by a resume is not allowed; any attempt to resume causes a new signal to be generated and propogated.

The second expression within a signal-call must evaluate to a uniword. Its value is the "message" associated with the signal; if absent, a value representing a null message is supplied. Within each catch phrase, the unique name and message are the values of the primaries "CODE" and "MESSAGE" respectively.

Binding mechanisms exist within the MPS support system for assigning the signal codes declared in other program-contexts to signal variables with the same identifiers declared in a given program-context.

```
resume ::= "RESUME" ['[ expr ']] ;
```

A resume statement specifies that propogation of a signal is to terminate and control is to return to the signaler. The optional expression must evaluate to a uniword, which becomes the value of the signal-call. A resume statement can appear only within a catch phrase.

Examples:

```
-- signal calls

   ERROR[SystemPunt, 7]

   SIGNAL[STRBoundsFault]

   SIGNAL[DataOverflow, n]

-- catch phrases

   mplUnwind:  -- clean-up
      ReleaseBlock[ptr];
   ANY:        -- count all other signals
      n ← n+1

   DataOverflow:
      BEGIN  v ← Allocate[MESSAGE];
      RESUME[v]       -- v is the value of the signal-call
      END
```

%Jump Statements%

jump-statement ::= ("EXIT" _ "REPEAT") .ID ;

A jump statement specifies that execution of a labeled statement is to be terminated or restarted. The identifier must stand as the label of a statement enclosing the jump statement, i.e., a label can be referenced in this way only within the (possibly compound) statement it precedes.

If "EXIT" appears, execution of the labeled statement is terminated. Processing continues as if the statement had terminated in the normal manner.

If "REPEAT" appears, execution of the labeled statement is restarted.

Note that jump statements are particularly useful within catch phrases and when so used provide the only transfers to nonlocal labels allowed in MPL (see the discussion of labeled statements).

Examples:

EXIT loop

REPEAT enter

%Bump Statements%

bump-statement ::= "BUMP" ["DOWN"] #<',>var;

A bump statement specifies the addition or subtraction of the constant 1 for each listed variable. The type of each such variable must have an associated length not exceeding the machine's word length. If "DOWN" appears, each variable is decremented; otherwise, each is incremented.

Examples:

BUMP DOWN n

BUMP i, j, k

%Conditional Statements%

conditional ::= if-statement _ case-statement ;

Conditional statements provide for conditional execution of a statement or, more generally, for selection from a list of alternative statements. There are two types of conditional statements, if-statements and case-statements.

If Statements

if-statement ::= "IF" expr "THEN" statement ["ELSE"

statement/;

In the execution of an if-statement, the expression following "IF" is evaluated. If the value of that expression is true (non-zero), the statement following the "THEN" is executed and the statement (if any) following the "ELSE" is skipped. If the value is false (zero), the statement following "THEN" is skipped and the statement (if any) following "ELSE" is executed. The expression must evaluate to a uniword.

Examples:

```
IF ptr = nil THEN RETURN

IF lineno = bm THEN EndPage[] ELSE lineno ← lineno + 1
```

Case Statements

```
case-statement ::= "CASE" sum

   ("OF" $case-of-stmt _ "IN" $case-in-stmt _ relational
$case-op-stmt)

   "ENDCASE" statement;

 case-of-stmt ::= #<',>(binrel _ sum) ': statement ';;

 case-in-stmt ::= #<',>(interval _ charclass) ': statement
';;

 case-op-stmt ::= #<',>sum ': statement ';;
```

The case statement provides a means of executing one statement out of many. The sum after the word "CASE" is evaluated, and the result, which must be a uniword, is saved to be used as the left operand of the relations labeling the various cases. Several relations may label a single statement. The statement following the first relation which is satisfied is executed. All other statements are skipped, even if relations labeling subsequent alternatives are satisfied. If none of the relations is satisfied, the (possibly null) statement following the word "ENDCASE" is executed.

In a case-of-stmt, each label specifies a list of items, and each item consists of a relational operator (or an implicit "=" if only a sum appears) and a right operand. The labeled statement is selected if any of the relations is satisfied.

In a case-in-stmt, each label specifies a list of intervals. The labeled statement is selected if the left operand falls within any of the intervals.

In a case-op-stmt, the relational operator is supplied with the left operand, and each label specifies a list of right operands.

Examples:

```
CASE char OF      -- select a scanning routine
   ' , EOL, TAB:                          -- ignored
      NULL;
   IN ['A .. 'Z], IN ['a .. 'z]:          -- letters
      ScanID[];
   IN ['0 .. '9]:                         -- digits
      ScanNumber[];
   '":                                    -- quote character
      ScanString[];
   ENDCASE                                -- all others
      CharacterToToken[char]

CASE x IN            -- assume 0 < v < r
   [-v .. v]:  match ← TRUE;              -- -v <= x <= v
   [-r .. r]:  NULL;                      -- -r <= x < v OR v < x
<= r
   ENDCASE  overflows ← overflows+1

CASE length-size <
   0:  ERROR;                             -- difference < 0
   minsplit: NULL;                        -- 0 <= difference <
minsplit
   ENDCASE SplitBlock[base, size]         -- minsplit <=
difference
```

%Iterative Statements%

do-statement ::= do-clause "DO" statement [do-test] [then-clause];

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

then-clause ::= "THEN" statement;

If the iteration is terminated because a do-test fails or the controlled variable reaches its bound, the statement in the then-clause is executed before control leaves the do-statement. If, however, the do-statement is left by other means, the statement in the then-clause is not executed.

do-clause ::= [for-clause] [do-test] ;

do-test = "WHILE" expr _ "UNTIL" expr ;

A do-test of the WHILE form succeeds if the value of the expression is TRUE (nonzero). The UNTIL form succeeds if the value of the expression is FALSE (zero). That value must be a uniword. There may be a single do-test 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 kinds of do-clauses provide for the specification, initialization, stepping, and testing of the do-statement's controlled variable.

for-clause ::=

"FOR" var ('← _ "FROM") expr ', expr [/"TO" expr/] _

[/"FOR" var/] thru-by-clause _

[/"FOR" var [/('← _ "FROM") expr/] _ "FROM" expr/
[/to-by-clause/] ;

The variable, if present, is used as the controlled variable, and it is reevaluated each time around the loop. Its type must be such that the associated length does not exceed the word length. All expressions must evaluate to uniwords.

In the first form of the for-clause, the value of the expression following the "←" or "FROM" is used to initialize the controlled variable. After each iteration, the expression following the "," is evaluated, and its value is assigned to that variable. If the "TO expr" is present, the expression following the "TO" is evaluated (once only) before the first iteration, and its value is saved. Iteration is terminated before execution of the controlled statement if the value newly assigned to the controlled variable is equal to the saved value.

This form offers an alternative to simply incrementing (or decrementing) the controlled variable and is particularly useful for following lists.

Example:

FOR ptr ← root, ptr.NextNode TO nil DO sum ← sum + ptr.NodeValue

-- the do-statement is terminated when the value of ptr, the controlled variable, equals nil.

The last two forms of the for-clause provide for iterative execution in which the controlled variable is stepped through an arithmetic sequence. Values in the sequence are determined by an initial value, a direction, an increment, and an optional limit.

thru-by-clause ::=

thru-clause [/by-clause/] _

```
   [by-clause] thru-clause ;

thru-clause ::= "THRU" interval ;

by-clause ::= ("UP" _ "DOWN") [expr] ;

to-by-clause ::=

   to-clause [by-clause] _

   by-clause [to-clause] ;

to-clause ::= "TO" expr ;
```

If no controlled variable is specified in the for-clause, an anonymous cell serves as that variable and, in the absence of the explicit initialization described below, has the initial value 0.

The possible directions are UP and DOWN. The default direction is UP, and the default increment is 1. If a by-clause appears, it specifies the direction; if the optional expression is present in that clause, the expression is evaluated (once only) before the iteration, and its value becomes the increment. The controlled variable is stepped after each iteration by adding the increment whenever the direction is UP and by subtracting it whenever the direction is DOWN.

A thru-clause specifies stepping the controlled variable through an interval, which is evaluated (once only) before the first iteration. When the direction is UP (DOWN), the controlled variable is initialized to the smallest (largest) integer in the interval, the controlled variable is stepped as described above, and iteration is terminated before execution of the controlled statement if the value newly assigned to the controlled variable prior to that execution is greater (less) than the largest (smallest) integer in the interval.

The expression following "←" or "FROM" specifies the initial value of the controlled variable when no thru-clause appears. In the absence of such initialization, the initial value of the controlled variable is unchanged by the for-clause. If neither a thru-clause nor a to-clause appears, the for-clause performs no test to terminate iteration. An expression following "TO" is evaluated (once only) before the first iteration, and its value is saved as the limit. When the direction is UP (DOWN), iteration is terminated before execution of the controlled statement if the value assigned to the controlled variable prior to that execution is greater (less) than the limit.

Examples:

```
-- the following FOR-statements are equivalent:

   FOR i FROM 0 TO n DO f[i];

   FOR i ← 0 UP 1 TO n DO f[i];

   FOR i ← 0, i+1 TO n+1 DO f[i];

   FOR i ← 0 UNTIL i>n DO f[i];

   FOR i THRU [0 .. n] DO f[i];

-- infinite loops:

   DO;    -- loops forever

   UP DO;  -- a slightly slower, but still infinite loop

   UNTIL FALSE DO;

   DO WHILE TRUE;

-- the following do-statement describes a bubble sort:

   FOR i DOWN THRU [1 .. u] DO
      BEGIN  sorted ← TRUE;
      FOR j THRU [1 .. i) DO
         IF a[j] > a[j+1] THEN
            BEGIN  sorted ← FALSE;
            a[j] ← a[j+1] ←: a[j];
            END;
      END UNTIL sorted
```

The following program section leaves i set to the index of the first element of the array a which is equal to x, or to -1 if no element equals x:

```
   (loop) FOR i THRU [0 .. LENGTH[A])
      DO BEGIN
         IF a[i]=x THEN  EXIT loop;
      END
      THEN i ← -1    -- if we fall out of the loop --
```

%Miscellaneous Statements%

```
misc-statement ::= inline _ system-call ;
```

These statements, designed specifically for MAXC and the PDP-10, are machine dependent and provide direct access to the underlying hardware and operating system (TENEX).

Inline Instructions

```
inline ::= "INLINE" '[ #<';>instruction '];

 instruction ::=
```

```
    [label]                          -- label for instruction
    .ID                              -- opcode
    [instcon ',]                     -- accumulator
    ['@]                             -- indirection
    ['= instcon _ .ID _ .NUM]        -- address
    ['( instcon ')];                 -- index

  instcon ::= .NUM _ .ID ;
```

An inline statement specifies execution of a sequence of machine instructions and allows direct access to the PDP-10 or MAXC hardware. The inline's brackets surround sequences of statements similar in form and meaning to PDP-10 MACRO statements, and each of these statements is assembled into a single machine instruction.

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

An identifier in an instcon must be a declared register (the address of which is used) or a defined constant (the value of which is used). In either case, the value is shifted left 23 bits (for the accumulator field), shifted left 18 bits (for the index field), or taken unchanged (for the address field or a literal).

The identifier of a variable or label declared in the enclosing procedure or program can be used in the address field, in which case appropriate indexing is automatically provided and the index field should be omitted. If the identifier of a variable declared within an INCLUDEd module is used, however, the index register must be specified explicitly.

Examples:

```
INLINE[ Mul a1,i ]      -- double length multiplication

INLINE[ Ildb r,bptr1; Idpb r,bptr2 ]    -- byte manipulation
```

System Calls

```
  system-call ::= "JSYS" '[ constant-expr ['', jsys-args]
$jsys-results '];
```

A system-call requests the services of the TENEX monitor. It is a direct mapping of the TENEX jump-to-system (JSYS) instruction into MPL. A JSYS accepts arguments and returns results in the registers beginning with register 1. Certain JSYS calls have multiple returns, and the significance of the returned values can differ for each possible return (see the TENEX JSYS manual).

The constant-expr specifies the JSYS number and selects the

service to be provided by the monitor.

jsys-args ::= $<',>expr;

Prior to calling the monitor, each argument expression is evaluated. Each value must be a uniword. These values are loaded into registers 1, 2, ..., with the order of the assigned registers identical to the order of the expressions.

jsys-results ::= ': $<',>[var];

The value of a system call is the relative location to which the JSYS returns (+1 for a normal return, +2 for a skip return, etc.). That value is used as an index to select a list of variables (possibly empty) from the jsys-results. There should be as many such lists as possible returns unless only a single return is possible. The values in registers 1, 2, ..., are assigned in order to the variables in the selected list. If no variable is specified for a particular register, that value is discarded.

Example:

JSYS[jsyscode,flags,v :errcode :,v]

-- the variables flags and v are loaded into registers 1 and 2, then the JSYS specified by jsyscode is executed. If the JSYS returns to the first location following the call, the value of the statement is 1 and the new value of register 1 is stored into errcode . If the JSYS returns to the calling location plus 2, the value of register 1 is discarded, the value of register 2 is stored into the variable v , and the value of the JSYS call is 2 --

%Null Statements%

null ::= ["NULL"] ;

The null statement specifies no action. It is provided as a convenience to the programmer.

%Compound Statements%

compound-statement ::= "BEGIN" $(block-dcl ';) $<';>statement "END" ;

A compound statement allows a sequence of statements to be bracketed and to appear in any syntactic position requiring a single statement. In the absence of explicit transfer- or jump-statements, the bracketed statements are executed in the sequence in which they appear.

block-dcl ::= "ENABLE" catch-phrase _ base-declaration ;

A catch-phrase following "ENABLE" is activated to process any signal generated or propogated as a result of executing a statement in the block headed by the "ENABLE". Execution of catch-phrases is described in connection with call-statements (q.v.). Multiple catch phrases at the head of a block are activated from right to left, and a signal generated by execution of such a catch-phrase is processed only by catch-phrases to its left.

The range of a base-declaration at the head of a block is the remainder of the block in which it appears (see MODULES).

Examples:

```
BEGIN  t ← i;  i ← j;  j ← t  END    -- exchange i, j

BEGIN
p ← Allocate[n];
   BEGIN
   ENABLE     -- deallocate on any kind of abnormal exit
      mplUnwind: Release[p, n];
   BASE DataArea ON ptr;    -- DataArea names an INCLUDE module
   -- arbitrary computation using the data area --
   Release[p, n];
   END    -- of the range of BASE and ENABLE
END
```

```
\ dlm ← -2*ilev

\ NewPage[]; dlm ← -ilev; justify ← FALSE
```

%APPENDIX: The Syntax of MPL(B)%

MODES

```
constant-expr ::= expr;

interval-constant ::=

   ('[_'() constant-expr '. '. constant-expr (')_']) ;

component-attribute ::=

   "INTEGER" _

   "IN" interval-constant _

   "FIELD" _

   "POINTER" ["TO" .ID] _

   "STRING" _

   .ID ;
```

```
field-attribute ::=

    component-attribute _

    "DESCRIPTOR" "FOR" "ARRAY" "OF" component-attribute _

    "PROCEDURE" [field-list] ["RETURNS" field-list] _

    "SIGNAL" ;

field-list ::=

    '[ (#<',> field-attribute _ #<',>( id-list ': field-attribute))
    '] ;

id-list ::= #<',> .ID ;

local-declaration ::= id-list ': local-attribute ;

local-attribute ::=

    "INTEGER" [expr-init] _

    "POINTER" ["TO" .ID] [expr-init] _

    "IN" interval-constant [expr-init] _

    "STRING" [ '[ expr '] _ '← expr ] _

    "FIELD" [expr-init] _

    .ID _

    "DESCRIPTOR" "FOR" "ARRAY" "OF" component-attribute [ '← expr ] _

    "ARRAY" ("OF" component-attribute array-init _
             interval-constant "OF" component-attribute) _

    "PROCEDURE" [field-list] ["RETURNS" field-list]
        [ '= procedure-body _ '← procedure-constant ] _

    "SIGNAL" [ "CODE" _ '← expr ] _

    "REGISTER" ['= constant-expr] _

    "TYPE" '= "RECORD" field-list ;

expr-init ::= ( '← expr _ '= constant-expr );

array-init ::= '← '[ $<',> expr '] ;
```

MODULES

```
body ::= $(locals ';) $(block-dcl ';) $<';>statement ;
program-module ::= .ID ': "PROGRAM" [field-list] '= prog-body '. ;
   prog-body ::= "BEGIN" [include-list ';] body "END" ;
data-module ::= .ID ': "DATA" [field-list] '= data-body '. ;
   data-body ::= "BEGIN" $(locals ';) "END" ;
procedure-body ::= "BEGIN" body "END" ;
locals ::= local-declaration _ base-declaration ;
include-list ::= "DIRECTORY" #<',> (.ID ': "INCLUDE" link) ;
   link ::= '( .ID ', .ID ', ') ;
base-declaration ::= "BASE" #<',> base-id ¤:
   base-id ::= .ID "ON" .ID ;
```

VARIABLE FORMS

```
var ::= .ID _ array-component _ record-component ;
array-component ::= var '[ expr '] ;
record-component ::=
  var qualifier _
  '( expr ') '. field ;
   qualifier ::= ('$ _ '.) field ;
   field ::= field-constant _ .ID ;
```

EXPRESSION FORMS

```
expr ::= conditional-expr _ disjunct ;
conditional-expr ::= if-expr _ case-expr ;
   if-expr ::= "IF" expr "THEN" expr "ELSE" expr ;
   case-expr ::=
     "CASE" sum
        ("OF" $case-of-expr _ "IN" $case-in-expr _ relational
```

```
        $case-op-expr)
        "ENDCASE" expr ;
      case-of-expr ::= #<',>(binrel _ sum) ': expr '; ;
      case-in-expr ::= #<',>(interval _ char-class) ': expr '; ;
      case-op-expr ::= #<',>sum ': expr '; ;
disjunct ::= #<"OR"> conjunct;
conjunct ::= #<"AND"> negation;
negation ::= ["NOT" _ '¤] relation;
relation ::= sum [binrel];
   binrel ::=
     ["NOT" _ '¤] ( relational  sum _ "IN" (interval _
     char-class));
   relational ::= '> '= _ '< '= _ '> _ '< _ '= _ '#;
   interval ::= ('( _ '[) expr '. '. expr ('] _ '));
   char-class ::=
     "CH" _ "ULD" _ "LLD" _ "LD" _ "UL" _ "LL" _ "L" _ "D" _ "PT" _
     "NP" ;
sum ::= #< '+ _ '- > prod;
prod ::= #< '* _ '/ _ "MOD" > bitor;
bitor ::=  #<"BITOR" _ "BITXOR"> bitand;
bitand ::= #<"BITAND"> factor;
factor ::= ['-] prim ;
prim ::=
  var _
  constant _
  function-call $qualifier _
  assignment-expr _
  '@ var _
  pseudo-function _
```

```
    rhs-statement _
    catchp-rhs _
    '( expr ') $qualifier ;
     pseudo-function ::=
       "FIELDMAX" '[ .ID '] _
       "RECORDSIZE" '[ .ID '] _
       "DISPLACEMENT" '[ .ID '] _
       "DESCRIPTOR" '[ .ID '] _
       "LENGTH" '[ var '] _
       "DIV" '[ prod ': var '] _
       ("MIN" _ "MAX") '[ #<',>expr '] ;
     rhs-statement ::= signal _ system-call ;
     catchp-rhs ::= "CODE" _ "MESSAGE" ;
  constant ::=
    literal _ string-constant _ procedure-constant _ field-constant _
    .ID ;
     literal ::= .NUM _ "TRUE" _ "FALSE" _ char ;
     char ::=
       .SR1 _
       "SP" _ "EOL" _ "ALT" _ "CR" _ "LF" _ "FF" _ "TAB" _ "EOS" ;
     string-constant ::= .SR ;
     procedure-constant ::= "PROCEDURE" .ID ;
     field-constant ::=
       .ID _
       '/ .ID _
       "FIELD" '[ [constant-expr ',] constant-expr ': constant-expr
       '] ;
  assignment-expr ::=
    var '← expr _ var '← ': expr ;
```

```
function-call ::= (.ID _ record-component) args ;
```

STATEMENT FORMS

```
statement ::= $label (simple-statement _ compound-statement) ;
simple-statement ::=
  assignment-statement _
  transfer-statement _
  jump-statement _
  bump-statement _
  conditional-statement _
  do-statement _
  misc-statement _
  null ;
assignment-statement ::=
  var '← expr _
  extractor '← (var _ function-call) ;
   extractor ::= '[ #<',>var '] ;
transfer-statement ::=
  call-statement _
  return-statement _
  stop-statement _
  signal-statement ;
   call-statement ::=
     "CALL" (.ID _ record-component) [args] _
     (.ID _ record-component) args ;
   args ::= '[ [$<',>expr] ['! catch-phrase ] '] ;
   catch-phrase ::= #<';> catch-case [; catch-all] _ catch-all ;
      catch-case ::= #<',> .ID ': statement;
```

```
      catch-all ::= "ANY" ': statement;
   return-statement ::= "RETURN" ['[ #<',>expr ']] ;
   stop-statement ::= "STOP" ;
   signal-statement ::= signal-call _ resume;
      signal-call ::=
        ("SIGNAL" _ "ERROR") '[ sig-list ];
      sig-list ::= expr [', expr] ['! catch-phrase];
      resume ::= "RESUME" ['[ expr ']];
jump-statement ::= ("EXIT" _ "REPEAT") .ID ;
bump-statement ::= "BUMP" ["DOWN"] #<',>var ;
conditional-statement ::= if-statement _ case-statement ;
  if-statement ::= "IF" expr "THEN" statement ["ELSE" statement];
  case-statement ::= "CASE" sum
     ("OF" $case-of-stmt _ "IN" $case-in-stmt _ relational
     $case-op-stmt)
     "ENDCASE" statement;
      case-of-stmt ::= #<',>(binrel _ sum) ': statement ';;
      case-in-stmt ::= #<',>(intrel _ charclass) ': statement ';;
      case-op-stmt ::= #<',>sum ': statement ';;
do-statement ::= do-clause "DO" statement [do-test] [then-clause];
   then-clause ::= "THEN" statement;
   do-clause ::= [for-clause] [do-test] ;
   do-test = "WHILE" expr _ "UNTIL" expr ;
   for-clause ::=
      "FOR" var ('← _ "FROM") expr ', expr  ["TO" expr] _
      ["FOR" var] thru-by-clause _
      ["FOR" var [('← _ "FROM") expr] _ "FROM" expr] [to-by-clause]
     ;
      thru-by-clause ::=
```

```
        thru-clause [by-clause] _
        [by-clause] thru-clause ;
      thru-clause ::= "THRU" interval ;
      by-clause ::= ("UP" _ "DOWN") [expr] ;
      to-by-clause ::=
        to-clause [by-clause] _
        by-clause [to-clause] ;
      to-clause ::= "TO" expr ;
misc-statement ::= inline _ system-call ;
   inline ::= "INLINE" '[ #<';>instruction '] ;
      instruction ::=
        [label]
        .ID [instcon ',] ['@] ['= instcon _ .ID  _ .NUM] ['(
        instcon ')] ;
      instcon ::= .NUM _ .ID ;
   system-call ::= "JSYS" '[ constant-expr [', jsys-args]
  $jsys-results '] ;
      jsys-args ::= $<',>expr ;
      jsys-results ::= $<',>[var] ;
null ::= ["NULL"] ;
compound-statement ::= "BEGIN" $(block-dcl ';)  $<';>statement
"END" ;
  block-dcl ::= "ENABLE" catch-phrase _ base-declaration ;
```