MPSDATA

by

James G. Mitchell

January 25, 1973

Xerox Palo Alto Research Center
3180 Porter Drive
Palo Alto, California 94304

Introduction 23

Declarations 25

declaration ::= "DECLARE" declist; 27

declist ::= $<',>(decitemlist [modexpr] [initialization]); 30

decitemlist ::= decitem / '(#<',>decitem '); 32

decitem ::= ID 34

If both the modexpr and the initialization are omitted, INTEGER is assumed; if the modexpr is elided but an initialization expression is given, the mode of the expression determines the mode of the variables in the decitemlist. 36 37 38

Examples: 40

abbreviated declaration	full declaration	
DECLARE a, b, c;	DECLARE a INTEGER, _ b INTEGER, _ c INTEGER;	42 44 46 48
DECLARE x = 3.75;	DECLARE x REAL = 3.75; % since 3.75 is a REAL constant	51
DECLARE (r, s, t) 4;	DECLARE (r,s,t) integer 4;	54

If both the modexpr and an initialization expression are given, the rules for mode agreement on assignment hold; these are discussed in detail below. 56 57

initialization ::= ('= / ') expression; 59

The form with an '= preceding implies that the item(s) being initialized will have their value frozen after the initialization is performed and may never appear as the left operand of an assignment or as the operand of an "address of" operator (see '$ below). The initialization will be performed as early as possible: it may even be accomplished when the declaration in which it appears is processed by the MPS compiler. Indeed, it is this facility which allows users to define new modes (or definitions of data structures) without incurring expensive runtime overhead. 61 62 63 64 65 66 67

Examples:

```
DECLARE a INTEGER = 5; % the variable a is equivalent to the integer value 5.

DECLARE b REAL 3.14; % the variable b will initially have the value 3.14, but it may be changed by assigning a different real value to it.

DECLARE (x,y) REAL 3.14; % declares x and y as real variables both initially to be assigned the value 3.14.

modexpr ::= modeconstant / modemacro / modexpr1;

modeconstant ::= basicmode / rowmode / builtinmode; % builtin modes are those which are defined as extensions but are always declared (note: this means that they could be overriden).
```

Basic Modes

```
basicmode = real / interval / set / indexmode / modemode;

indexmode ::= integermode / elementmode;

real ::= "REAL"; % a floating point value.
```

Intervals

```
interval ::= "INTERVAL";
```

An interval is a pair of integers and represents the set of integer values in that range. An interval constant has the following syntax:

```
intervalcon ::= ( '[ / '( ) expression ".." expression ( ') / '] );
```

The parentheses and brackets have the normal mathematical meaning of open and closed interval ends, respectively.

Examples:

```
[-2 .. 5]   % includes -2, -1, 0, 1, 2, 3, 4

(0 .. n]    % includes 1, 2, ..., n.

[5 .. 4]    % includes nothing; an empty interval
```

Interval values consist of a pair of integers, which correspond to the endpoints of the closed interval, and may be compared for quality and containment. Equality is expressed using "="; the relational operator "IN" is used to express containment.

E.g.

```
[-2 .. 5] IN [-1 .. 6) = TRUE

[-2 .. 5] IN [-1 .. 6] = FALSE

[-2 .. 4) IN [-2 .. 4) = TRUE
```

The length of an interval is obtained by the builtin function

LENGTH;

```
LENGTH (interval) = UPPER(interval) - LOWER(interval);
```

[Note: This means the interval [5 .. 3] has length -2; this could be normalized to -1 for all empty intervals but that does not seem necessary.]

Integers

```
integermode ::= "INTEGER";
```

A simple integer variable i, declared as

```
DECLARE i INTEGER;
```

is an integer variable which may range over the interval [-2*5 .. 2**35).

Sets

```
set ::= "SET" (intervalexpression / integerexpression / '(
$<',>ID ') );
```

Sets are closely related to the notion of interval; a variable s whose mode is SET[-5 .. 5] may take as values subsets of the set -5, -4, -3, ..., 3, 4, 5 . Such a set variable is equivalent to a vector of bits, one for each possible elemental value of the set; for example,

```
S  {-5, -3, -1, 1, 4, 5};
```

would result in s being equivalent to the bit string 101010100011.

If an integerexpression is the option used, it is equivalent to a set over the interval [0 .. integerpression).

If a set is defined by giving a list of identifiers in the set definition, a number of things occur:

a) In scopes where elements of the set are required, any of the listed identifiers may be used; they are the constant values of the elements of the set.

b) There is a complete order defined on the identifiers which corresponds to their order in the list.

The operations on set values include UNICN, INTERSECT, complement (NOT) and the relations <, =< (or "IN"), =, ≠, >=, >, meaning inclusion.

Set Constants

```
setcon ::= '{ $<',>elementexpr '} "NULLSET" / setmode identifier;
  %this will probably have to be changed for NLS since it gets
  upset at seeing "{" or "}".

{}=NULLSET in the empty set.

elementexpr ::= integerexpr / ID;
```

Integer element values may only be used iff the set was defined over an interval, and ID's may be used iff the set was defined by an identifier list.

Example:

```
DECLARE s1 SET[0 .. 7];
s1   {7, 4, 3, 1};
DECLARE s2 SET(a, b, c, d, e);
set2   {b, a, d};
```

There is a simple conversion between a set defined over an interval and the interval, and vice versa. Thus,

```
s1   [4 .. 6];
```

is allowed and is equivalent to

```
s1   {4, 5, 6};
```

If a variable of mode set has been declared, its name may be used as a set constant as the equivalent of the entire set.

Example:

```
DECLARE set1 MODE=SET(a, b, c, d);
DECLARE s set1;
s   set1; % is valid and is equivalent to %
s     {a, b, c, d};
```

Set Elements

setelement ::= "ELEMENT" "OF" setexpression;

A set element has a mode which is associated closely with a set-mode: the element may take as values the elements of the set. If the set is defined over an interval, a set element may take any integer value in the interval as its value. If the set is defined by a list of identifiers, those identifiers are the only constant values which may be assigned to an element variable associated with that set.

Examples:

```
DECLARE Color ELEMENT OF SET (Red, Blue, Green, Yellow,
Purple, Orange);
```

Color is then an element variable for which the following are all valid expressions:

```
Color   Red;

Color = Red % boolean expression

Color < Blue % order is Red < Blue < Green < Yellow < Purple
< Orange.
```

However, arbitrary arithmetic on element variables defined for a set of identifiers is not allowed.

A set element may be converted to a singleton set value when used in combination with set values:

Example:

```
DECLARE Primaries SET(Red, Blue, Green);
DECLARE hue Primaries;
DECLARE(c1, c2) ELEMENT OF Primaries;
c1   Red;
```

```
    hue   c1; % c1 considered as singleton set   Red .                255
    (c1=<hue)=TRUE % since c1=Red and hue= Red                        257
```

There are also a number of operations available for sequencing 259
through the elements contained in a set variable. The operation 260
MIN(set) yields the first element contained in the set.

```
  E.g.  hue   Blue;                                                   262
        MIN(hue)=Blue % is then true.                                 264
```

The operation FROM(set) gives the MIN element of the set and 266
deletes the element from the set.

MAX(set) yields the last element of the set. 268

```
  implies that   hue   hue UNION Green; LAST(hue)=Green,              270
```

assuming the above statements have been executed. 272

Of course, a set s may have as value the empty set, NULLSET, in 274
which case

```
  MIN(s) = MAX(s) = NIL                                               276
```

There is also an option on the MPS FOR statement for sequencing 278
THRU a set, element by element:

```
  FOR c1 THRU hue DO ...;                                             280
```

will assign the values Blue and Green successively to c1 given 282
the above assignments to hue. If the expression following the 283
work "THRU" yields the complete set of a given set-mode, then the 284
iteration is over all the possible elements of the interval or
over the elements of the set.

```
  mode ::= "MODE";                                                    286

     Examples:                                                        289

        DECLARE  integer MODE = INTEGER; % the variable "integer"     292
        is another way of saying "INTEGER".

           DECLARE  (a,  b) integer; % Semantically equivalent to     295
           "DECLARE (a, b) INTEGER;".
```

A variable of mode MODE is a pointer to a data structure called 298
a ModeDescriptor. If the MODE variable is only assigned a 299
value at compile time and is not altered by the program being
compiled, then all mode checking for variables declared to be 300

of that mode can be done at compile time and involves no run time overhead.

References and Pointers

ref ::= "REF";

A variable of mode REF can refer to any object in MPS; internally, a REF carries a handle on a MODE and a hardware "address" (probably a byte pointer on the PDP10). It is expected that routines which manipulate the MPS runtime environment (such as CREATE, the binder, and the debugger) will be the primary users of objects of this mode. Attempting arbitrary operations on a REF object will probably not be allowed, or if it is may carry a large expense for interpretation.

The only loophole in the typed data structure facilities is the function MAKEREF which takes a pointer to a mode a pointer to any object and makes a REF value for that object. The storage allocation may use this facility, for instance.

pointer ::= "POINTER" '(modexpr '); % modexpr must be avaluable at compile time.

A variable which is declared to be a PTR(amode) may take as value the generalized address of an object of mode amode. A pointer may be used to gain access to an object by the dereferencing operation:

'[ptrexpression ']

Example:

```
DECLARE (am, bm) amode;

DECLARE ptrvar POINTER (amode);
```

Then [ptrvar] has mode amode and

```
ptrvar   @am;
```

assigns the address of am to ptrvar. Until the value of ptrvar is changed, [ptrvar]=am is true thereafter.

The assignments
```
bm   am;
_bm  [ptrvar];
```
are valid and semantically equivalent. However, the statement

```
_bm   ptrvar;                                                    345
```

is invalid, since bm is not a pointer, and since no automatic dereferencing of pointers is performed in MPS! 347 348

A pointer must be able to reference any "nameable" object, including a 1 bit field in a word: 350 351

it thus must contain as much information as a PDP-10 byte pointer: word address, position of object within the word, and the width of the field in which the object resides. 353 354

A REF value to an object of mode M may be coerced to a POINTER(M) value and stored in an object of mode POINTER(M). 356 357

Initialization of Objects 359

initialization ::= ('= / '¬) expression; 361

If the '¬ option is used to initialize a declared variable, the expression will be evaluated whenever a new instance of the variable is created (usually at runtime). For a local variable of a procedure, for instance, it will happen on each new call on the procedure. 363 364 365

The modes specified in a declaration become available in the environment as soon as they appear and may be used to initialize the variable(s) being declared: 367 368

E.g., 370

```
DECLARE Color IN SET(Red, Green, Blue)   Blue;                   372
```

is valid since the set {Red, Green, Blue} is known by the time the initialization expression is compiled. 374 375

The variables being declared in a declist, however, are not known until the decitem in which they appear has been completely processed. Thus, if var were a variable declared in some outer scope, the declaration 377 379

```
DECLARE var REAL   1+var;                                        381
```

is valid and declares an instance of var in the scope of the declaration. Whenever this instance of var is created, it will be initialized to one plus the value of the outer var. 383 384

Composition of Modes 385

Modes may be composed in a number of ways to create new modes. These compositions take four forms: painted modes, row modes, structured modes, and variant modes.

Painting a mode produces a new mode whose undertype is the mode before painting; this serves to distinguish between modes which are structurally equivalent but which are intended to be semantically distinct.

Row modes represent indexable, homogeneous aggregates of objects; this includes the notion of an array, as in Algol 60, for instance.

Structured modes are the result of composing a set of modes by concatenation; unlike row modes, the components may have different modes.

Variant modes allow one to describe a mode which includes more than one possible alternative form and definition; in any given instance of an object whose mode is variant, the alternative mode which it possesses at a given moment is determinable.

Conceptually, all extended modes are created as the result of calling MODE valued procedures. An object of mode MODE in MPS is in reality a pointer to a class of objects called ModeDescriptors (i.e, there is a mode ModeDescriptor). The question of how such procedures are declared and used will be delayed until discussed in more generality below.

Painted Modes

Normally, two mode declarations which are syntactically identical will map into the same mode (i.e., into a pointer to the same ModeDescriptor). In order to distinguish between a pair of REALs which are to be viewed as a complex number and a pair of REALs viewed as Cartesian coordinates, for instance, one may paint the first with the identifier "complex" and the latter with the identifier "Cartesian", thusly:

```
DECLARE complex MODE=complex :: RECORD(x REAL, y REAL);
DECLARE Coordpair MODE=Cartesian :: RECORD(REAL, y:REAL);
DECLARE (c1, c2) complex;
DECLARE (point1, point2) coordpair;
```

Then, MODE(c1)≠MODE(point1) because the paint hides their identical substructures. The paint is associated with the ModeDescriptor and not with the MODE variable per se. However, the submode of c1 is "RECORD(x:REAL, y:REAL)" and is equivalent to the submode of point1.

To include painting in the general MODE syntax, replace the rule

```
modexpr ::= modeconstant / modemacro / modexpr1,
```

by the rule

```
modexpr ::= [paint] (modeconstant / modemacro / modexpr1);
```

and add the rule

```
paint ::= ID "::";
```

Row Modes

A row mode describes a collection of objects of the same mode which can be indexed by the elements of a set. A row is defined by given the mode of the components of the row and the set which specifies the number of elements in the row and the mode of the values which may be used to select (index) a component for the row.

There are two row modes:

```
rowmode ::= arraymode / sequencemode;

arraymode    ::=    ARRAY[indexexpression]rowtail;    % an
indexexpression is an expression of mode indexmode.

    indexmode ::= setmode / integermode;

    rowtail ::= "OF" modexpr;
```

The indexexpression, if present, need not be evaluable at compile-time. The mode given by the indexexpression is called the domain of the array and the mode of the elements of the array is called its range. If the indexexpression is not present, the mode produced by the declaration does not describe an array, but rather an ArrayDescriptor. An ArrayDescriptor has two parts, a length and a pointer value: the length is the number of elements in the array whose first element resides at the address to which the pointer refers. There is not a single mode called ArrayDescriptor, but one for each different elemental mode over which an array is declared. Array descriptors greatly facilitate the passing of arrays as arguments to procedures since they carry length information along with them; moreover, since the length is in the ArrayDescriptor and is not attached to the array itself, subarrays can be accessed by modifying the array descriptors

printer and length information. This is done using the builtin function SUBARRAY (ArrayDescriptor, interval) which returns an ArrayDescriptor for a subarray of the array defined by the ArrayDescriptor parameter. That subarray consists of the elements of the original whose indices begin in the given interval.

The primary operation specific to a row is selection of components: an element of an array is selected by specifying an element of the set which defines the domain of the array.

The syntax for selection is:

```
arrayselector                ::=          arrayexpression
'[ indexelementexpression' ];
```

The mode of the indexelementexpression must correspond to the mode of the domain of the arrayexpression.

Examples:

```
DECLARE monthtable:  ARRAY set(Jan, Feb, Mar, Apr, May,
                     June, July, Aug, Sept, Oct, Nov, Dec)
                     OF ELEMENT OF SET [1 .. 31]
```

defines an array of 12 elements which is indexed by the element constants Jan, Feb, ..., Nov, Dec; each array element is an element of the set defined over the interval [1 .. 31]. An element of monthtable may be selected only by one of the identifiers Jan, Feb, ..., Nov, Dec.

```
DECLARE localarray   ARRAY LENGTH(parameterarray) OF REAL;
```

defines an array of REALS whose length (i.e., number of elements) is to be the same as that of some other array named "parameterarray". In order to select an element of this array, the index value must be of mode

```
ELEMENT OF SET [0 .. LENGTH(parameterarray) )
```

However, the declaration for an element of a set requires that the set expression be evaluable at compile time, which is not the case in this example.

The INDEXMODE declaration corrects this problem:

```
modexpr ::= "INDEXMODE" '( Arraymode ');
```

Example:

```
    DECLARE month INDEXMODE(monthtable);                              512

  means "month" has mode                                              514

    ELEMENT OF SET (Jan, Feb, Mar, Apr, May, June, July, Aug,         516
                    Sept, Oct, Nov, Dec);                             517
```

If the arraymode is in fact an ArrayDescriptor mode, then the INDEXMODE declarator can be used in a procedure to declare variables which can be used in selecting elements from the array parameter. 519 520

The range of an array may itself be an array: 522

```
    DECLARE matrix ARRAY[1 .. n] OF ARRAY [1 .. n] OF REAL;           524
```

This declares matrix as a square matrix, i.e., an array each of whose components is an array of real values. Selecting the jth element in the i'th row is denoted by 526 527

```
    matrix[i][j]                                                      529
```

Of course, matrix[i] is an array of reals and may be treated as an object in its own right. 531 532

Sequences 534

```
  sequencemode ::= "SEQUENCE" rowtail;                                536
```

The length of a sequence is determined when an instance of one is created and is stored at the beginning of the vector of elements. A sequence may be indexed by an integer and a check is made to determine that the index value is in the range [0 .. LENGTH(sequence)). Unlike arrays, the elements of a sequence may not themselves be sequences (although they can be arrays). 538 540 542

Examples: 544

```
    DECLARE simplestring SEQUENCE OF ASCII                            546
```

(ASCII is a pre-defined set whose constants are written as 'a, b', etc., and whose internal representaion corresponds with the ASCII character set.) 548 549

```
    DECLARE era SEQUENCE OF monthtable;                               551
```

An era is a row whose length, at compile time, is unresolved; each component of an era is a 12 component array of months as declared previously. 553 554

Records

```
record ::= "RECORD" '( declist ');
```

A record is a composite mode whose (generally nonhomogeneous) components are (selector, mode) pairs. A selector is an identifier which can be used to access or set a component of a record object. For example,

```
DECLARE pair RECORD (x INTEGER, y INTEGER);
```

This declares a record object whose name is "pair". Its components are a pair of integers. To select one of the components, the binary selection operator "$" is used: e.g.,

```
pair$x    1;
```

stores the value 1 in the "first" component (named "x") of "pair"; and

```
pair$y    pair$x;
```

copies the value in the first component into the second.

If a program has a pointer to a record, and is to select a component from the record pointed to, a "." is used in place of the "$" operator:

```
DECLARE ptrpair POINTER(pair);

ptrpair   @pair;  % store pointer to the variable pair as
                    the value of ptrpair.
```

This means that

```
ptrpair.x=pair$x
```

and

```
ptrpair.y=pair$y
```

The "." operator is thus equivalent to the composition of dereferencing the pointer value and "$" selection on the resultant object:

```
i.e.  ptrpair.x = [ptrpair]$x
```

Default initializations for record components may be specified just as for normal variables. Whenever a record object is created, initializations are performed (in a left to right

order) either from a creation time value, or a declaration time value of the former is not supplied.

Examples:

```
DECLARE mixed RECORD (s1 REAL   3.14, s2 INTEGER = 5);
```

This means that mixed$s1=3.14 and mixed$s2=5, and furthermore that mixed$s2 is subsequently unalterable, due to the "=" form of initialization.

RECORD is in actuality a function which takes a single parameter, which is an unevaluated list, and returns as value a mode (i.e., a pointer to a ModeDescriptor) which describes the record mode. The record mode descriptor in turn refers to a set mode (which contains the selector identifiers as the elements of the set) and to an array whose domain is the selector set and whose range is the mode SELECTOR. The elements of this array are used in selecting components from objects of the record mode. Of course, the array may not even necessarily survive to be a runtime object if that is not necessary.

Selectors are closely connected with the notion of pointers touched upon earlier. In its simplest form, a selector is a record object of a predefined mode and describes a component of some record as a relative bit address and a width (in bits); it might be represented as a byte pointer on a PDP-10, for instance.

A selector may also be a function which maps a pointer value onto another pointer value. Thus, if fs is a functional selector and rec a record for which fs is a valid selector, then

```
rec$fs
```

as a righthand side value is equivalent to

```
[fs(@rec)];
```

i.e., to calling fs with a pointer to rec. The pointer value returned by fs is then "dereferenced" to yield the required object. A selector variable is declared by

```
selectormode ::= "SELECTOR" '( modexpr ');
```

and a variable so defined may assume only selector values which yield values of mode POINTER(modexpr).

Example:

```
DECLARE listcell MODE = RECORD(value INTEGER, link
POINTER(listcell));

DECLARE linkfield SELECTOR(POINTER(listcell));
```

The variable linkfield may take as value a selector for the link component of a listcell. The binary operator @ is used to generate selector values:

```
selectorvalue ::= recordmodexpression '@ selectoridentifier;
```

The value of the expression

```
listcell@link
```

for instance, is a selector value which could be assigned to linkfield and used in selector operations.

```
I.e., linkfield  listcell@link;
```

is a valid MPS statement. After it is executed, linkfield may validly and meaningfully be used wherever the constant selector link is used.

JGM: It looks as though this compromises the MPS mode machinery unless the assignment to linkfield also includes the mode listcell, so that any selection operation in which linkfield is subsequently used can check that linkfield's value is a selector for the object from which component selection is being done.

Since RECORD is a mode-valued function, it may be used wherever a modexpr is allowed: this includes using it to initialize a MODE variable.

Examples:

```
DECLARE complexnum RECORD(realpart REAL, imagpart REAL);
```

declares "complexnum" as a record object. If one wanted to declare a number of such objects without rewriting the call on RECORD, he could write:

```
DECLARE complex MODE = RECORD(realpart REAL, imagpart REAL);

DECLARE a complex;

DECLARE (c1, c2) complex;
```

In this case, the "=" initialization of the MODE variable "complex" allows it to be used to declare objects of that mode. Such initialization is the primary means of defining data modes without incurring runtime overhead for dealing with objects of the extended mode.

Variant Modes

A mode may be defined as a "discriminated union" of a number of modes. A variable which has a variant mode may take on values of several different modes, a feature useful for defining parts of records which may contain one of a number of different types of values.

```
union ::= simpleunion / functionalunion;
```

A simpleunion is a pair consisting of a union of previously declared modes and a tag which is an element value. The tag associates specific modes in the union with tag values; and a list of declarations prefixed by tag values specifies a mapping from tag values to modes. Functional unions are semantically similar except that the mapping which associates possible modes with tag values for a union is a function.

```
simpleunion ::= "UNION" tag  setmodexpr uniontail;

tag ::= ID;

uniontail ::= "OF" variantcases "END";

variantcases ::= $<';> singlevariant

singlevariant ::= $<',>elementexpression ':  declist;
```

Examples:

```
DECLARE LispCell MODE = UNION CellType SET(Atom,
Number, Cons)

   OF  Atom:  Name SEQUENCE (ASCII);

      Number Value INTEGER;

      Cons:  (Car, Cdr) POINTER(LispCell)

   END;
```

Thus a LispCell really has two main components: a tag named CellType, which is a set and probably occupies two bits; and a

variant part which is a SEQUENCE of 7-bit BYTES, an INTEGER, or a pair of POINTERS to objects of mode LispCell.

Instead of including a specific tag as part of a variant mode, one may specify an implicit functional tag. This function must accept a pointer to an object of the variant mode and must return an element value which signifies which variant of the set of possible modes currently resides in the object.

The syntax for declaring such a functional union is similar to the definition for simpleunions:

```
functionalunion ::= "UNION" "CALL" functionID['':setmodexpr] uniontai

functionID ::= ID; % which is a procedure which takes a
POINTER to an object of the UNION mode and returns an
element of the set specified by the setmodexpr.
```

The ':setmodexpr phrase may be omitted if there is a definition of functionID available which itself specifies the set over which the return values may range.

Example: a LispCell in which a tag field was not necessary could be declared as

```
DECLARE LispCellx MODE =
   UNION CALL CellModeFn ELEMENT OF(Atom, Number, Cons)
   OF
      Atom:    Name SEQUENCE(ASCII);
      Number:  Value INTEGER;
      Cons:    (Car, Cdr) POINTER(LispCellx)
   END;
```

CellModeFn must be a function which takes a POINTER (LispCellx) object as argument and returns one of the element values Atom, Number, or Cons, depending on the mode of the value "resident" in the object.

Creation and Destruction of Objects

MPS allows convenient syntactic short forms for writing constants of builtin modes:

```
builtinconst ::= realcon / intervalcon / integercon /
modecon / arraycon / sequencecon / setcon;

realcon ::= realnumpart ['E decimalint];

   realnumpart ::= [decimalint]'.decimalpart /
```

```
   decimalint['.[decimalpart]];                                          786

   decimalpart ::= decimalint;                                           788

   Examples:  1E5, 1., .1, 1.7, 1.7E0                                    790

intervalcon  ::= ( '[ / '( ) expression ".." expression ( ') /           793
'] );

   Examples:  [-5 .. 7), [0 .. n), (0 .. n]                              795

integercon ::= octalint / decimalint;                                    797

   octalint ::= #('0/'1/'2/'3/'4/'5/'6/'7)'B;                            799

      Examples:  172B, 111B                                              801

   decimalint ::= #('0/'1/'2/'3/'4/'5/'6/'7/'8/'9);                      803

      Examples:  101, 509, 123456789                                     805

modecon ::= modeconstant; % described earlier                            808

   Examples:  INTEGER, REAL, BYTE(3), BYTE, MODE, ARRAY [0               810
   .. 7) OF REAL, ARRAY [0 .. 7) OF ARRAY [0 .. 7) OF MODE               811

arraycon ::= ;                                                           813

   There are no simple constants whose mode is ARRAY; there              816
   are, however, constants which are sequences, and a
   mapping function from sequence objects to array objects:              817

sequencecon ::= '( #<',>expression ');                                   819

   Examples:  (1, 2, 3, 4), (a+b, 7, 9, a-b)                             822

   Iteration of constants (to more flexibly initialize                   824
   sequences) is achieved in two ways: the first involves                825
   sequence-valued procedures, and the second repetition of
   sequences.

   There are a set of built-in functions which take an                   827
   INTEGER and sometimes a mode as well and produce a
   sequence.  For example, the builtin function                          828
   IOTA(n:INTEGER) returns as value the sequence (1, 2, ..., 
   n).  For those acquainted with Iverson's APL, this may                829
   sound familiar.  If such a result is to become part of a              830
   sequence, it must be unbundled and concatenated into it;
   e.g.,
```

(5, IOTA(4)9, 9) is equivalent to (5, (1, 2, 3, 4), 9), which is not a valid sequence constant because the modes of the elements differ (the first element is an integer and the second a sequence of integers). The sequence primitive UNSEQ performs this function. If the ', between elements in a sequence constant is thought of as a concatenation operator, then

(5, UNSEQ(IOTA(4)), 9) is seen to be (5, 1, 2, 3, 4, 9), which is what is needed.

```
setcon ::= ID / integercon; % either of which must be a valid element of a set
```

Objects of extended modes can also be created. The syntax for a "constructor" is

```
constructor ::= [modexpr] ' $<',>consitem ');

consitem ::= [ selectorpart expression;

   selectorpart ::= [ (decitemlist / setexpression / intervalexpression) ': ];
```

Examples:

```
DECLARE complex MODE = RECORD(realpart REAL, imagpart REAL);
```

The following are equivalent constructors for the complex number 5+i:

```
complex (5,1)
complex (imagpart:1, realpart:5)
complex ((imagpart,realpart):(1,5))
complex (realpart:5, 1)
```

Using selectors to specify values for the components of an object allows one to list them in any order he pleases. Selectors may also be elided and the component values written in a list; or some selectors may be given and other component values listed. The rules governing this are the following:

Recall that the selector identifiers in a record declaration form a set associated with that record mode. They are ordered in the same way as the elements of a set mode. If no selectorpart is specified for a consitem in a constructor, the selector to be associated with that value is the successor of the selector for the previous consitem

in the list. If this is the first consitem, the first element of the selector set is assumed. Thus,

```
complex(realpart:5, 1)
```

is equivalent to

```
complex(realpart:5, imagpart: 1).
```

since imagpart is the successor of realpart in the selector set for complex.

Elements of a row mode can be initialized by specifying a subset or interval (depending on the set which is the domain of the row) in place of a selector:

Example:

```
DECLARE introw ARRAY[ 0 .. 7 ] OF INTEGER;

    introw  ([ 0 .. 3]:(1, 2, 1, 2), (3 .. 7]:(10, 10, 10, 10, 10));
```

will set introw to be the array (1, 2, 1, 2, 10, 10, 10, 10, 10).

The repetition of values (like the five 10's above) can be eliminated by writing a scalar value which will be replicated as necessary to satisfy the needs of the set-valued selector. Then the previous example may be written as:

```
introw  ([0 .. 3]:(1, 2, 1, 2), [3 .. 7]:10);
```

More generally, the values in a sequence construct will be replicated (in a round-robin) fashion until the selector set is satisfied. For example,

```
introw  ([ 0 .. 3]:(1, 2), (3:7]:10);
```

is equivalent to the previous example: the sequence (1, 2) is completely replicated twice to satisfy [0 .. 3]. A less elegant, but equally correct statement is

```
introw  ([0 .. 0]:1, [1 .. 3]:(2, 1), (3 .. 7]:10);
```

In this case, the elements of (2, 1) are replicated to produce [2, 1, 2).

If it is clear from the context what mode a constructor represents, the normally preceding modexpr may be omitted: e.g.,

```
DECLARE (c1, c2) complex;
c1   complex:(5, 1); % explicit mode
c2   (5, 1); % implied mode is the mode of CL
```

The situation, of course, is complicated in more general expressions, depending on the associativity of operators, etc. (More about this later.)

A constructor is an expression form and thus may appear as one of the expressions in a constructor:

Example:

```
DECLARE cplxpair RECORD((c1, c2) complex);

DECLARE cx complex   complex(2,3);
```

Then

```
cplxpair   ( complex(5, 7), complex(2, 3));
```

is valid and equivalent to the following:

```
cplxpair     (  (5, 7), (2, 3)); % mode of expressions is
desirable.

cplxpair   (c2:cx, c1:(5, 7)); % cx is a valid expression
of mode complex.
```

Constructions for objects of variant mode may be written in the same way as those for non-variant modes. However, when a value is assigned to such an object, the tag field (if it is a simpleunion) is set to the appropriate element value as a side effect of the assignment. If it is necessary to specify which variant is meant, the set element naming the variant may be specified.

Example:

```
DECLARE consex MODE =

   UNION tag:IN SET(r1, r2, cplx)
   OF
      r1:   real REAL;
      r2:   real REAL;
      cplx: imag complex
```

```
      END;                                                          977

      DECLARE rrc consex;                                           979
```

The values may be assigned to rrc as in the following: 981

```
      rrc   r1(3.5);                                                983

      rrc     complex(4,  5); % also sets tag  cplx; it is not      987
      necessary to specify the tag of the variant in this case
      since only one alternative has mode complex.
```

Assignment and Extraction 989

Extraction is the dual of construction. It is often 992
desirable to be able to extract the values of a number of
components of an object: e.g., to "explode" the object and 993
store some of its component values into simple variables for
further operation.

```
      lhs ::= ... / '( extractorlist ') ;                           995
      extractorlist ::=#<;>extractoritem ;                          997
      extractoritem ::= selectorpart lhs ;                          999
```

Example: 1001

```
      DECLARE  consex:MODE = RECORD ( s1: INTEGER, s2: ARRAY [0     1003
      .. 3]) ; _                                                    1004
      DECLARE (c1, c2) consex;                                      1006
      DECLARE (a, b, c, d, e) INTEGER ;                             1008
      C1     :  ( s1:   3  ,  ( [0 .. 4] : ( 1, 2, 3, 4) ) ; %     1010
      construction _                                                1011

         Now c1$s1=3 and c1$52[0]=1, etc. _                         1014
```

Now an extraction: 1016

```
      [52:(a, b, c, d), s1:e)   c1; % extraction.                   1018
```

This assignment statement is equivalent to the following 1020
sequence of simple assignments: _ 1021

```
      a   c1$s2[0];                                                 1023
      b   c1$s2[1];                                                 1025
      c   c1$s2[2];                                                 1027
      d   c1$s2[3];                                                 1029

      c   c1$s1;                                                    1031
```

The following are also valid and are all equivalent to the example just given:

```
{ s2:{[3, 2, 1, 0] : ( d, c, b, a) , s1:e   c1; _
{ e : ((a, b, c, d))   c1;
{ e s2[0]:  a, s2[1]:b, s2[2]:c, s2[3]:d )   c1;
```

As with constructions, array elements are one level lower than the array itself as a component; the first two examples above show this.

Traditionally, a procedure accepts a list of zero or more arguments and may return a list of zero or more results. An MPS procedure can accept either zero or one argument and may return either zero or one value. In both directions, the object (if any) which is transmitted is considered to be a record and may therefore have components of any valid MPS mode:

i.e., any definable object may be sent as a parameter to or returned from a procedure.

Variable members of parameters may be sent using SEQUENCES; different types of parameters may be sent as variant objects; and the value returned by a function may be dissembled or manipulated in the same way as any other MPS object. Indeed, the procedure is considered to be the creation function for its return value; in order to make this association unique, the mode of the return record is painted with the name of the procedure which returns it.

The syntax for a procedure declaration is

```
proceduremode ::= "PROCEDURE" sendreturnlist;

sendreturnlist      ::=      [sendrecordmode      ["//"
returnrecordmode][;

sendrecordmode  ::= '(declist') / recordmodexpr % which
must evaluate at compile time to a recordmode %;

returnrecordmode  ::= '(declist') / recordmodexpr % which
must also yield a record mode at compile time %;
```

The record sent to a procedure is constructed by the caller just prior to calling the procedure: this construction may take advantage of the syntax and mechanisms available for constructors. Parameters may be specified by selector name instead of strictly positionally; or parameters may be omitted and defaults (as specified by the sendrecordmode

specification) supplied. Similarly, return records may be dissembled using extractor mechanisms. For example:

```
DECLARE fortyfive PROCEDURE(X REAL) // (X REAL, Y REAL);
```

Then, if one only wants to look at the Y value returned by fortyfive, it can be done thusly:

```
IF fortyfive(3.5)$Y ≠ 3.5 THEN RETURN;
```

I.e., the procedure call "fortyfive(3.5)" creates an object of mode

```
fortyfive :: (X:REAL, Y:REAL)
```

to which the selector Y may validly be applied.

Examples:

```
DECLARE metric PROCEDURE(x:REAL  0, y:REAL  0, z:REAL
0) // (length:REAL);
```

The following are all valid calls on metric; the version on the right is the equivalent, completely specified call:

```
metric(3.5, 4)          metric(x:3.5, y:4, z:0)

metric(Y:4)             metric(x:0, y:4, z:0)

metric(5)               metric(x:5, y:0, z:0)

metric(3.5, 4, 1.7)     metric(x:3.5, y:4, z:1.7)
```

Those calls which elide some arguments act as projection functions with the specific projection specified by the PROCEDURE declaration.

If the procedure is to correspond to some actual procedure this can be indicated by initializing the procedure object (metric, for instance, is a procedure "variable": nothing in the declaration associates it with any explicit piece of program). Assume that metric were an actual procedure and one wanted to define a "projection" of metric for 2-spare called hypotenuse:

```
DECLARE hypotenuse PROCEDURE(x;REAL, y:REAL, z:REAL = 0)
// (length:REAL) = metric;
```

Since the z parameter is fixed by the declaration, hypotenuse may only be called with the x and y parameters

specified. As well, hypotenuse will be initialized at compile time to be equivalent to the procedure metric if the procedure metric is then accessible. Otherwise, the MPS binding facilities will attempt to bind hypotenuse to a suitable procedure named metric using the context available at binding time (which is later than compile time and before executing a call on the procedure).

Ports and signals are similarly declared:

```
portmode ::= "PORT" sendreturnlist;

signalmode ::= "SIGNAL" sendreturnlist;
```

The sendrecordmode specifies what objects may be sent out over a port and the returnrecordmode specifies the form of replies over the port. A signal actually causes a two component record to be sent to any catch phrase which is called to handle the signal: the first component identifies the signal and the second is the sendrecord. A RESUME from a signal must transmit an object of mode returnrecordmode to the signalling program.

Macros

User-defined Operations for Objects of Non-basic Modes

Code Generation

A model for semantic expressions:

Assign a standard, published name to each operator or builtin function in MPS.

IF an operator meets operand(s) with which it is not prepared to deal, a search may be started for a generic macro, signal, port, or procedure definition for that case.

E.g., a+b where a is an integer and b is of mode complex, say, might result in some control form such as

```
PLUS(INTEGER, COMPLEX)
```

If an appropriate generic macro is found, a and b are substituted for the macro's formals and then the macro body (a parse tree) is processed semantically (see Macros for a more detailed description of this process). If a suitable generic procedure, port, or signal declaration

is found, the parse tree is altered to a form such as ComplexAdd(a,b).

Suppose the operation is selection ('. or '$) and the expression appears on the left hand side of an assignment statement.

E.g., b."RealPart"←a;

Then, a routine is needed which has the form

Select(complex, STRING,Boolean)

actually, a mode which covers STRING will do

where Boolean = TRUE left hand evaluation
Boolean = FALSE Right Hand Evaluation

Select must return a value of mode PTR (modex) if Boolean = TRUE; if Boolean = FALSE, it may return a value of mode modex or PTR(modex).

In general, perhaps there ought to be a boolean left hand side or right hand side argument to any user-selected functions or macros for operators.

The foregoing assumes that routines will be called with argument lists and not mode pointers (since the actual call will occur at runtime). To cast the PROCEDURES ProcedureMode and RecordMode in this mold implies interpretation of function calls at compile time (probably prompted by the '= initialization), assuming that the procedures exist in the compiler's environment.

It is an open question whether a procedure appearing in an included module is considered to exist in the compiler's environment. Macro definitions in INCLUDE modules certainly do.

```
selectorpart ::= [selectorexpr '"];                                    119
```