



PM – Programming Parallel Models

Language Reference – Preliminary Draft (incomplete)

Tim Bellerby, University of Hull, UK

Contents

Background	3
Conventions used to define language syntax	3
Lexical Elements.....	3
Modular Structure.....	5
Types, values and objects	6
Numeric types	6
Non-numeric types	7
Structures and records.....	8
Polymorphic Types.....	8
Optional Types	9
Arrays	9
Variables, Constants and References.....	9
Procedures	12
Expressions.....	14
Statements.....	15
Parallel execution.....	16
Syntax.....	18

Background

PM (Programming Parallel Models) is a new open source programming language designed for implementing environmental models, particularly in a research context where ease of coding and performance of the implementation are both essential but frequently conflicting requirements. PM is designed to allow a natural coding style, including familiar forms such as loops over array elements. Language elements have been included (or more importantly excluded) from the language in a careful combination that enables the language implementation to both vectorise and parallelise the code. The following goals guided the development of the PM language:

- Language structures facilitate the generation of vectorised and parallelised code
- Vectorisation and parallelisation should be explicit and configurable
- Programmers should be able to concentrate on coding the algorithm – not on how to get it to run efficiently using whole-array operations and/or in parallel
- Support for data structures required in modelling: matrices, vectors, grids (including grids of matrices, vectors and sub-grids) and meshes with generalised connectivity.
- PM code should be as readable as possible by those not familiar with the language.
- PM should be formally specified – not defined by an implementation.

Conventions used to define language syntax

PM syntax will be described using the following extended BNF notation:

<i>name</i> ::= <i>list</i>	Define a non-terminal element in terms of other elements
<i>name</i>	Non terminal element
dim	Keyword
'>='	Character combination
[<i>elements</i>]	Elements are optional – may appear zero or one times
{ <i>elements</i> }	Elements may appear zero, one or more times
<i>e1</i> <i>e2</i>	Exactly one of the listed element sequences must be present
(<i>e1</i> <i>e2</i>)	Brackets may be used to group selections that are not enclosed by { } or []

Lexical Elements

Comments start with a '!' and continue to the end of the line

```
! Comments are ignored
```

PM uses upper and lower case letters, digits and a number of special symbols:

```
letter ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'  
         'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'  
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The following single characters and character combinations are used as **delimiters**:

```
'$'  '&'  '('  ')'  '**'  '*'  ';'  '//'  '/'  '=>'  ':'  '||'  ';'  '@'  '['  
']'  '_'  '{'  '|'  '}'  '-'  '"'  '+'  '<'  '<='  '='  '=='  '>'  '>='  '%'  
'..'  '!'  '->'  '::'
```

To cater for environments with restricted character sets, the following substitutions are always permitted:

```
'/' for '['      '/' for ']'      '(:' for '{'      ':)' for '}'      '%%' for '@'      ':/' for '|'  
':/ for '||'
```

White space characters (space, newline, form feed and horizontal tab) may appear between lexical elements (delimiters, names, keywords, numeric constants) but not within them. Names, keywords and numbers must be separated from each other by white space. Otherwise white space is optional.

New lines are not generally significant, but in any context where a semicolon ';' is expected, the semicolon may be omitted if the following lexical (non-white-space) symbol starts on a new line.

Names have a maximum length of 100 characters. They are comprised of upper and lower case letters, decimal digits, and the underscore character '_'. They may not start with a digit. Letter case is significant.

```
stem    ::= letter{'_'|letter|digit}
name    ::= ['_']name
cname   ::= '_name'_
```

Names that start with '_' or '\$_' are unique to the module within which they are defined. A module entity defined using such a name may not be referred to from another module.

The following are **keywords** and may not be used as names:

and	any	array	arg	argc	
by	case	check	const	create	
debug	dim	div	do	else	elseif
enddebug	enddo	enddo	endfor	endif	
endproc	endselect		endwhile		false
for	if		in	is	let
	on	or	opt	otherwise	over
proc	prod	reduce	ref	repeat	require
result	select	seq	then	true	type
until	use	using	var	where	while

Numeric constants take the following forms:

```
number      ::= [ integer_constant | float_constant | imaginary_constant ] [ 's' | bits ]
integer_constant ::= [ ['+' | '-' ] ] { digit } 'r' [ { [ digit | letter ] } ]
float_constant  ::= [ ['+' | '-' ] ] { digit } [ '.' digit { digit } ] [ 'e' [ [ '+' | '-' ] ] digit { digit } ]
imaginary_constant ::= [ integer_constant | float_constant ] 'i'
bits           ::= '_' { digit }
```

String constants are enclosed by float quotes"" and may include any character other than "". They may not span more than one line.

```
"Hello World"
```

Modular Structure

A PM program consists of one or more source files, each corresponding to a **module**. A module defines a collection of **procedures**, **types** and **parameters** (module level constants). A **program module** also contains executable statements. **Interface modules** specify the interface (API) that must be implemented by an actual model. Module names are constructed from a sequence of standard PM names optionally joined by ‘.’ delimiters. They are linked to source file names in an implementation-dependant manner. Module names starting with **lib.** refer to standard libraries.

```
module ::= program_module | library_module
program_module ::=
    [ decls ] statements
library_module ::=
    decls [ debug statements enddebug ]
modname ::=
    name { '.' name }
```

Names may optionally start with an underscore character. Such names are local to the module in which they are used and do not match names in any other module into which they are imported.

```
decls ::=
    { import ';' } ( import | decl ) { ';' decl }
import ::=
    use modname [ modifiers ] { ',' modname [ modifiers ] }
```

Modules may include other modules using an **use** statement. In its simplest form, **use module_name** the statement imports all definitions in the named module into the current module. Type and parameter declarations should not conflict with other definitions in the module containing the import statement – whether directly defined or imported from other modules. Imported procedure definitions merge with definitions imported to or declared in the module containing the include statement. This merging joins definitions across a PM program – if two different modules define a procedure with the same name, the merging of their definitions affects both the module imported to and both imported modules.

```
modifiers ::=
    '{' modifier { ',' modifier } '}'
modifier ::=
    [ type | param | proc ] name '=>' name
```

Name clashes may be avoided by using a qualifying clause in the **use** statement. This lists specific elements to import and optionally renames them:

```
use model4 {
    type model_params
    param theta => model_theta
    test => test_model
}
```

The optional => operator in this context renames the given element. If an element in the list is not qualified as **type**, **param**, or **proc** then it is assumed to be a procedure (**proc**).

Types, procedures and parameters occupy separate name spaces – they may have the same name as each other.

Type, procedure and parameter declarations follow any include statements in the module. The simplest of these is the **param** declaration which simply defines a named constant:

```
paramdec::=
    param name '=' xexpr
```

For example:

```
param pi = 3.14159265
```

param statements may refer to each other, irrespective of the order in which they are defined. However, such references must not be recursive or mutually recursive.

Type and procedure declarations are described later in this document.

In a program module, the (optional) declarations are followed by executable statements. Such a module may be run as a program. It may not, however, be included in other modules. Library modules may be included in other modules. They may not contain executable statements after the declarations except for an optional single debug statement that should contain module testing code. The PM system may optionally check that the library module debug statement references all procedures and types in the module –directly or indirectly - and issue a warning if this is not the case. Interface modules are described later in this document.

Types, values and objects

PM programs operate on *values* stored in *objects*. A *type* defines a (possibly conceptually infinite) set of values and is constructed using a *type expression*. type may consist of a set containing only a single value. However, it remains distinct from that value. A *conformance* relationship is defined between types and between values and types:

- A value V conforms to an abstract type T if $V \in T$
- An abstract type T conforms to abstract type T if $T \subseteq U$

Numeric types

PM supports a range of integer types, defined in the following table. The definitions of these types are flexible to enable portability of code. Code requiring, e.g., a strict bit size should use inquiry functions to check that a given type meets the expected requirements.

int	<i>short integer</i>	<i>System defined standard integer</i>	
long	<i>long integer</i>	<i>Integer capable of counting elements in larges possible array</i>	
int8	<i>8-bit integer</i>	<i>OR smallest integer holding ≥2 decimal digits</i>	<i>OR same as long</i>
int16	<i>16-bit integer</i>	<i>OR smallest integer holding ≥4 decimal digits</i>	<i>OR same as int8</i>
int32	<i>32-bit integer</i>	<i>OR smallest integer holding ≥9 decimal digits</i>	<i>OR same as int16</i>
int64	<i>64-bit integer</i>	<i>OR smallest integer holding ≥18 decimal digits</i>	<i>OR same as int32</i>
int128	<i>128-bit integer</i>	<i>OR smallest integer holding ≥36 decimal digits</i>	<i>OR same as int64</i>

Integer constants may be defined with respect to any base between 2 and 62 by specifying the base followed by 'r' and then the digits, with 'a'..'z' and 'A'..'Z' representing 10 to 36 respectfully (case insensitive).

```
123          2r101011101    16rdeadbeef
```

By default integer constants yield values conforming to **int**. Long integer constants should append an **L** to the value. For other integer types, append an underscore and the corresponding (assumed) number of bits:

```
889874324897284746      ! long
255_8                    ! int8
2r10101010101010101010101010101010_32 ! int32
```

PM also defines a set of real types:

real	Standard (system defined) floating point value		
double	Standard (system defined) double precision floating point value		
real32	32-bit floating point	OR real number with ≥ 1 decimal digits in mantissa	OR same as real
real64	64-bit floating point	OR real number with ≥ 15 decimal digits in mantissa	OR same as real32
real128	128-bit floating point	OR real number with ≥ 24 decimal digits in mantissa	OR same as real64

Real constants must contain a decimal point and may contain an exponent preceded by 'e'.

```
-5.2      2e3      4.2e-20
```

By default real constants yield values conforming to type **real**. To specify a **double** constant, append a **d**. For other real types append an underscore and the assumed number of bits:

```
-5.2d      ! double precision
3.2e-3     ! real
1.2e-2_32  ! real32
```

Complex types are defined as follows:

cpx	Complex number formed from real components	
double_cpx	Complex number formed from double components	
cpx64	64-bit complex number	OR complex number based on real32 components
cpx128	128-bit complex number	OR complex number based on real64 components
cpx256	256-bit complex number	OR complex number based on real128 components

There are no direct complex constants. **Imaginary constants** are float constants terminating with the letter 'i'. They yield a complex constant of a corresponding type:

```
3.0i      ! cpx
1.0_32i   ! cpx64
-2.2-30di ! double_cpx
```

Non-numeric types

The **bool** type contains the logical values **true** and **false**

The **string** type contains values that are arbitrary length character strings. **String constants** are enclosed by double quotes "" and may include any character other than "". They may not span more than one line.

```
"Hello World"
```

The **name** type contains values that are valid PM names. Name constants are formed by preceding a name by the dollar symbol:

```
$x
```

The **proc** type contains values that are valid PM procedure names (including operator symbol). Name constants are formed by preceding a name by the **proc** keyword:

```
proc cell_model
proc +
```

Structures and records

Structures and records provide a mechanism to create aggregate values. They are very similar, except that it is not possible to alter a single component of a record, while this is possible for a structure.

A structure or record values is created using a generating expression:

<code>[struct rec] [<i>name</i>] '{ <i>name</i> '=' <i>exp</i> { ';' <i>name</i> '=' <i>exp</i> } '}'</code>
--

For example:

```
struct { name="John Smith", age=42 }
rec point { x=2, y=4 }
```

A structure or record value tags each component with a name. The whole value may optionally tagged with a further name. Component names are local to the structure or value. The name tagging the whole structure or record resides in a name space that is global to the program (unless preceded by an `'_'` in which case it is local to the given module)

A structure or record type is constructed using:

<code>[struct rec] [<i>name</i>] '{ <i>name</i> [':' <i>type</i>] { ',' <i>name</i> [':' <i>type</i>] } '}'</code>
--

For example:

```
struct { name:string, age:int }
rec point { x:int, y:int }
```

A structure or record value is a member of a given structure or record type if all of the following apply:

1. The value is a record and the type is defined using **rec** or the value is a structure and the type **struct**
2. The whole-value name matches the corresponding name given in the **struct/rec** type, or both are absent
3. The same component names are present (not necessarily in the same order)
4. If a type constraint is associated with a component name in the type expression then the corresponding component of the structure or record value, associated with that name, conforms to that type constraint.

A given component of a structure or record may be accessed using the `'.'` operator. Structure components may be modified using the same operator.

```
person:=struct { name="John Smith", age=42 }
location:=rec point { x=2, y=4 }
person.age=person.age+1
print(person.age)
print(location.x)
```

Polymorphic types

A polymorphic value has an internal representation that is capable of representing any value in a given type. A value of a given polymorphic type is created using a generating expression:

any [*type*] '{ *expr* }'

If a type is not given then it is set equal to the concrete type of the expression (see section on variables below). The value contained within a polymorphic value is obtained using the unary '*' operator.

For example:

```
type int_or_string is int,real
a:= any int_or_string{ 1 }
a= any { 'Hello' }
print(*a)
```

A polymorphic type expression has the following form:

any '{ [*type*] }'

A polymorphic value conforms to type **any{T}** if the contained value conforms to *T*.

Optional types

An optional value has an internal representation that is capable of representing any value in a given type or the special value **null**.

opt [*type*] '{ *expr* }'

The value contained within an optional value is obtained using the unary '*' operator.

An optional type expression has the following form:

opt '{ [*type*] }'

Arrays

An array type is a generic type indicating the storage of a set of values over corresponding to elements of a given domain. Array values may be created using the generating expression:

array '{ *expr* ',' *expr* }'

However most PM code will use the binary **dim** operator, e.g.: `0.0 dim grid(0..3,0..3)`. The operator creates a new array by replicating the element value over every point in the given domain. The second argument to the **dim** operator must conform to domain.

Array types are defined using:

type '[' *typelist* ']

Variables, Constants and References

Objects are created and associated with names using a declaration statement:

```
x := 4
const message = "Hello World"
```

An object has a *concrete type* that defines the set of values it is capable of storing. The concrete type of an object is determined by as follows:

1. If the initial value conforms to any of: **int long int8 int16 int32 int64 int128 real double real32 real64 real128 cpx double_cpx cpx64 cpx128 cpx256 bool string name proc type** then then that type is the concrete type of the object.
2. If the initial value is a structure or record, then the concrete type is a structure or record type with component type constraints given by the concrete types of the corresponding components, determined recursively.
3. If the initial value is a polymorphic value created using **any** $T\{expr\}$ then the concrete type is given is a polymorphic type **any** $\{T\}$
4. If the initial value is an array, then the concrete type is an array type with element and domain types equal to concrete types obtained recursively from corresponding components of the array value.

If the name is declared using **var** or **':='**, values in the object may be changed later in the program. If it is declared using **const** then no changes to its value are permitted. The restriction for **const** names is implemented by forbidding them from appearing on the left hand side of an assignment statement or as a reference argument to a procedure. A name is defined until the end of the block of statements in which it is defined. Local (var, const and reference - defined later) names may not local shadow names defined in enclosing blocks or clash with procedure parameter names.

The values stored in variables may change over time. Values are changed in one of two ways: (i) using an assignment statement and (ii) by passing the variable as a reference argument to a procedure.

```
var x = 4      ! Value stored in x is 4
x = x + 1     ! Value changed to 5
float(&x)     ! Value changed to 10
              ! - assuming procedure float has been defined
```

Some procedures return more than one value. In this case, multiple left hand sides are permitted:

```
assignment ::=
  lhs '=' expr
  lhs ',' lhs { ',' lhs } '=' call
  name '->' name qualifier
lhs ::=
  name qualifier | '_'
qualifier ::=
  { '.' name | '[' exprlist ']' | '{' exprlist '}' }
```

For example:

```
x, y, _, z = returns_four_args(a)
```

An underscore may be used as a place filler, allowing a given returned value to be ignored. It is also possible for the left hand side of an assignment to refer to a component of an object.

```
a.f = b
a[3, 2] = c
```

A reference declaration assigns a name directly to an existing object, without creating a new object first. A reference is both created and changed using the same statement form:

<i>assignment ::=</i> <i>name '->' name qualifier</i>

For example:

```
r -> x  
r -> y.f  
r -> a[3..9]
```

A reference declaration has the same scope as a variable or constant declaration.

DRAFT

Procedures

A procedure defines an operation on a set of objects, which may change some of their stored values. Procedures may also return one or more values. A procedure declaration defines the name of the procedure, parameters on which it operates and their associated type constraints and any values returned.

```
proc square(x) = x**2

proc calc_stats(data: float dim any) do
  .....
  result=mean,std_dev
endproc
```

Procedure names may be simple names (with or without a leading underscore) or may be one of the following operators:

**+ - div mod * / ** == /= > >= and or // => dim = [] {} opt
.. <.. ..< <..< by**

Note the absence of < or <= from this list – these operators are always defined relative to > or >= respectfully.

Each procedure is associated with a **signature** consisting of the following:

1. The name of the procedure
2. The number of values the procedure will return.
3. The number of arguments the procedure will accept
4. The type constraints on the procedure's parameters
5. Which parameters are declared to be reference parameters

Keyword parameters (defined below) are not part of the signature.

A procedure signature *P* is said to be **strictly more specific** than another procedure signature *Q* if:

1. The procedure names are the same.
2. The numbers of parameters are the same (allowing for variable argument definitions in either procedure).
3. All reference parameters occur in the same position.
4. The type constraint for each parameter of *P* conforms to the type constraint for the equivalent parameter of *Q*
5. Type constraints are not strictly equal to one another for at least one parameter position.

A procedure call invokes a procedure, supplying it with a list of values and/or objects to operate on. If the call is part of an assignment statement (discussed in detail below) then the call also provides a list of objects to receive any values generated or declarations to receive created objects.

```
u, var v = myproc(x, &y, z*2)
```

If a procedure call is nested inside another procedure call, then the nested call is assumed to return a single object to the definition of an anonymous intermediate constant (again see the discussion below for details on declarations). Thus the following are equivalent:

```
y = f(g(x))

const _anon = g(x)
y = f(_anon)
```

Reference parameters, denoted by **&**, indicate that the procedure may change the value stored in corresponding object. Any procedure call must precede the corresponding argument with an **&**.

When encountering a procedure call, PM finds all procedures with signatures that **match** the call. A call matches a procedure signature if :

1. The procedure name is the same.
2. The number of arguments equals the number of parameters defined by the signature, or is greater than or equal to the number of parameters for a variable arguments signature.
3. All reference parameters are associated with a reference argument, starting with an **&**, in the same position.
4. The object supplied for each argument conforms to the type constraint for the corresponding parameter.

If more than one procedure matches a given call then PM will try to find a procedure that is strictly more specific than all the other candidate procedures. If no such procedure exists, then the call is **ambiguous** and an error condition arises.

Keyword arguments, if present, are not considered when finding a matching procedure. However, if the matching procedure will not accept a supplied keyword parameter then an error condition arises.

A procedure may accept a variable number of arguments:

```
proc set_numbers(&dest:int dim any, arg:int...) do ....
```

From the body of the procedure, the excess arguments are accessed using **arg[]** and **argc**:

```
for i in 1..argc do
  dest[i] = arg[i]
endfor
```

It is also possible to pass optional arguments *en masse* to a procedure call:

```
proc process(a) do
  print("Value is"//a)
enproc
proc process(a, arg...) do
  process(a)
  process(arg...)
endproc
process_values(1,2,3,4)
```

Arguments passed in this way do form part of the signature of the call – they are used in the matching process and do not necessarily have to correspond to the variable argument portion of the called procedures parameter list.

Keyword parameters provide optional values to the procedure call. They are specified by providing a default value. Keyword parameters follow all other parameters and **arg ...** (if present). Keyword parameters do not form a part of the signature of the procedure. An error results if the procedure selected by signature matching is not able to receive the keyword arguments provided in the call.

```
proc run_model(params : model_params, iterations = 1000,
  relaxation = 0.02) do
  .....
endproc

run_model(my_param_set, relaxation=0.042)
```

It is possible to define a procedure that takes additional unknown keyword parameters and passes these on to other procedures. This is achieved using the **key** keyword:

```
proc process_opts (&optarray, first_opt=.false, second_opt=33, key...) do
  ...
  process_other_opts (key...)
endproc
```

These additional keyword arguments may only be passed to another procedure – there is no facility to inspect them individually.

Expressions

An expression consists of a set of nested procedure calls. The usual infix notation is provided for common mathematical operations, but this is *syntactic sugar* for procedure calls.

Operation	Prior-ity	Equivalent call	Description
<code>x[a,b]</code>	1	<code>proc [] (x,a,b)</code>	Array subscript
<code>-x</code>	2	<code>proc - (x)</code>	Minus
<code>x**y</code>	3	<code>proc ** (x,y)</code>	Power
<code>x*y</code>	4	<code>proc * (x,y)</code>	Multiplication (including matrix multiplication)
<code>x/y</code>	4	<code>proc / (x,y)</code>	Division (float result)
<code>x div y</code>	5	<code>proc div (x,y)</code>	Division (integer result)
<code>x mod y</code>	5	<code>proc mod (x,y)</code>	Remainder
<code>x + y</code>	6	<code>proc + (x,y)</code>	Addition
<code>x - y</code>	6	<code>proc - (x,y)</code>	Subtraction
<code>x == y</code>	7	<code>proc == (x,y)</code>	Equals
<code>x /= y</code>	7	<code>proc /= (x,y)</code>	Not equal
<code>x > y</code>	7	<code>proc > (x,y)</code>	Greater than
<code>x >= y</code>	7	<code>proc >= (x,y)</code>	Greater than or equal
<code>x < y</code>	7	<code>proc > (y,x)</code>	Less than
<code>x <= y</code>	7	<code>proc >= (y,x)</code>	Less than or equal
<code>x and y</code>	8	<code>proc and (x,y)</code>	Logical and
<code>x or y</code>	9	<code>proc or (x,y)</code>	Logical or
<code>x // y</code>	10	<code>proc // (x,y)</code>	String concatenation
<code>x .. y</code>	11	<code>proc .. (x,y)</code>	Range creation (discrete ranges only)
<code>x .. y by z</code>	11	<code>proc .. (x,y,z)</code>	Range creation
<code>x => y</code>	12	<code>proc => (x,y)</code>	If x is type then y else null optional value
<code>x y</code>	13	<code>proc (x,y)</code>	If x is not null (or null optional value) then x else y
<code>x => y z</code>	13	<code>proc (proc=>(x,y), z)</code>	If x is true then y else z
<code>x dim y</code>	14	<code>proc dim(x,y)</code>	Array creation – spread copies of x over domain y
<code>opt x</code>	15	<code>proc opt(x)</code>	Create object of optional type

Sub-expressions may be separated from the main expression using **where**.

```
s = a * -exp (b/a) where a = c/sqrt(b)
```

There may be multiple values defined after a where keyword. These clauses may not refer to each other, but it is possible to follow with a second where statement and associated clauses:

```
a = x **2 / y**3 where x = s/p, y=s/q where s = sqrt(p**2 + q**2)
```

Statements

PM provides a range of conventional control statements:

```
if xexpr then statements { elseif xexpr then statements } [ else statements ] endif  
select xexpr { ( case xexprlist | default ) do statements } endselect  
while xexpr do statements endwhile  
repeat statements until xexpr
```

For example:

```
if x > y then print("X is greater") endif  
  
if x>y then  
    print("X is greater")  
else  
    print("Y is greater or equal")  
endif  
  
select digit  
    case '1','3','5','7','9' do print("Odd")  
    case '2','4','6','8' do print("Even")  
    otherwise do print("?")  
endselect  
  
repeat  
    x = receive(channel4)  
until x == 0  
  
while x>100 do  
    x = x/2  
endwhile
```

These statements have conventional interpretations. Conditional expressions must return a **bool** value. A runtime error will result if they do not. The select statement tests for equality using the standard '==' comparison.

Two statements allow for debugging and may optionally not be compiled and executed. The **check** statement confirms that an expression yields a true value and raises an error if this is not the case. The **debug** statement executes a block of code only if debugging is enabled – with an optional string descriptor for finer control of which **debug** statements are executed:

```
check [string!:] xexpr  
debug [string!:] statements enddebug
```

For example:

```
check value /= null  
  
debug "loops": print("Entering main loop") enddebug
```

Parallel execution

The **for** statement executes its body in parallel for every element of a domain or array:

```
for iter subexp [ seq | [ using assignlist ] ] loopbody endfor
iter::=
    name { 'name' } in exp { 'exp' }
    name from exp follow qualifier
loopbody::=
    do statements
    find statements [ otherwise statements ]
```

For example:

```
for pixel in image do
    pixel = min(pixel, threshold)
endfor
```

Statements in the body of a parallel loop are not normally allowed to modify variables defined outside of the scope of the **for** statement. It is also possible to require that the loop body is executed sequentially.

```
for s in array seq do
    sum = sum + f(s)
endfor
```

Scientific models will usually require interaction between loop invocations. PM enables this using the '@' operator. Used as a unary operator (@x) this provides an array view of a given expression across all invocations. Used as a binary operator (x@ndb_desc) it returns values from a local neighbourhood of the calling invocation. For example:

```
for cell in model_array do
    advection=advection_model(cell)
    advected_cell=cell@advection
    cell=model(advected_cell,parameters)
endfor
```

In the latter case, the result will be an array with an optional type and off-edge values undefined.

```
for pixel in image do
    pixel = median(pixel@grid(-1..1,-1..1))
endfor
```

The '::' operator repeatedly applies a given binary operation applied between the value of a given expression across all invocations of the enclosing parallel loop:

```
for cell in model_grid do
    cell=model(cell,parameters)
    cell=cell/sum::cell      ! Normalise
enddo
```


A reduction operation must be defined using a specialised procedure definition:

```
proc sum:: (x,y) = x + y
```

The binary operation is applied with arbitrary ordering and bracketing. This will only give a deterministic result if the operation is both commutative and associative.

It is also possible to define procedures incorporating '@' and '::' operators that can only be called inside parallel for statements:

```
proc @mean_filter(x,n)=sum(nbd)/count(nbd)
  where ndb=grid(-n..n,-n..n)
for pixel in image do
  pixel = @mean_filter(x,1)
endfor
```

To ensure synchronisation, an '@' operator, '@' procedure call or '::' operator may not occur inside a conditional statement (if, select, while, repeat) or a sequential for statement whose iteration domain is local to the enclosing parallel for loop.

The version of the **for** statement described above always invokes the loop body for every element of the domain. A second version executes arbitrary invocations of the loop body until at least one such body encounters a **found** statement:

found <i>assignlist</i>

Out of the loop invocations that encounter the found statement, only one will execute the statement (or pass on its side effects). If all loop invocations complete without encountering a found statement, then the statements in the **otherwise** clause (if present) are executed.

```
for pixel in image find
  if pixel>threshold then
    found high_pixel=pixel
  endif
otherwise
  high_pixel=-1
endfor
```

Syntax

module::= *program_module* | *library_module*

program_module::=

[*decls*] *statements*

library_module::=

decls [**debug** *statements* **enddebug**]

decls::=

{ *import* ';' } (*import* | *decl*) { ';' *decl* }

import::=

use *modname* [*modifiers*] { ',' *modname* [*modifiers*] }

modname::=

name { '.' *name* }

modifiers::=

{ '{' *modifier* { ',' *modifier* } '}' }

modifier::=

[**type** | **param** | **proc**] *name* '=>' *name*

decl::=

procdecl | *typeddecl* | *paramdecl*

signature::=

procsig | *typesig* | *paramsig*

procdecl::=

proc *procname* *params* '=' *xexprlist* [**do** *statements* **endproc**]

proc *procname* *params* [**check** *exprlist*] **do** *statements* [**result** '=' *xexprlist*] **endproc**

procname::=

[*name* | *op* | *name* '::' | '@' *name*]

op::=

'+' | '-' | **div** | **mod** | '*' | '/' | '**' | '==' | '/=' | '>' | '>=' | **and** | **or** | '//' | '=>' | '||' | **dim** | '=' | '[' | '{' | **opt** |
'..' | '<..' | '..<' | '<..<' | **by**

params::=

'(' { *param* ',' } [*param* [',' *keyparams*] | **arg** '...' [',' *keyparams*] | *keyparams*] ')'

param::=

['&'] *name* [':' *type*]

keyparams::=

{ *name* '=' *expr* ',' } (*name* '=' *expr* | **key** '...')

namelist::=

name { ',' *name* }

typeddecl::=

type *name* ['{' *namelist* '}'] [**in** *typelist*] (**is** *typelist* | **includes** *typelist* | **also includes** *typelist*)

paramdecl::=

param *name* '=' *xexpr*

type::=

name ['{' *typelist* '}']

type '[' *typelist* ']'

opt '{' *type* '}'

any '{' [*type*] '}'

[**struct** | **rec**] [*name*] '{' *name* [':' *type*] { ',' *name* [':' *type*] } '}'

statements::=

statement { ';' *statement* } [';']

statement::=

if *xexpr* **then** *statements* { **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**

select *xexpr* { (**case** *xexprlist* | **default**) **do** *statements* } **endselect**

while *xexpr* **do** *statements* **endwhile**

```

repeat statements until xexpr
for iter subexp [ seq | [ using assignlist ] ] loopbody endfor
check [string':'] xexpr
debug [string':'] statements enddebug
found assignlist
definition
let assignlist
assignment
call [ closure ]

iter::=
  name { ',' name } in exp { ',' exp }
  name from exp follow qualifier

loopbody::=
  do statements
  find statements [ otherwise statements ]

definition ::=
  const assignlist
  ( name | '_' ) { ',' ( name | '_' ) } ':= ' xexpr

assignlist::=
  assignment { ',' assignment } subexp

assignment ::=
  lhs := expr
  lhs ',' lhs { ',' lhs } := call
  name '->' name qualifier

lhs::=
  name qualifier | '_'

expr::=
  lowest to highest precedence
  expr dim expr
  expr '||' expr
  expr '=>' expr
  expr [ '..' | '<..' | '..<' | '<..<' | by ] expr
  expr '//' expr
  expr or expr
  expr and expr
  not expr
  expr [ '==' | '/=' | '>' | '<' | '>=' | '<=' | in | includes ] expr
  expr [ '+' | '-' ] expr
  expr mod expr
  expr [ div | prod ] expr
  expr [ '*' | '/' ] expr
  expr '**' expr
  '-' term
  opt term
  term qualifier

subexp::=
  [ check exprlist ] { where namelist := expr { ',' namelist := expr } }

xexpr::=
  expr subexp

xexprlist::=
  exprlist subexp

exprlist ::=
  expr { ',' expr }

call::=

```

```

    name('arglist ')
    proc procname '('arglist')'
term ::=
    subterm
    value
    '@' name [ '(' arglist ')' ]
    subterm '@' subterm
subterm ::=
    any type '{' subterm '}'
    array '{' expr ',' expr '}'
    name
    '('expr')'
    call
    array
    [ struct | rec ] [ name ] '{' name '=' exp { ';' name '=' exp } '}'
value ::=
    constant
    arg '[' expr ']' | argc | lo | hi | true | false
    proc procname | type type
constant ::=
    number | string | '$' name
arglist ::=
    { arg ',' } ( arg | arg '...' ) [ ',' keyargs ]
    [ keyargs ]
arg ::=
    '&' name qualifier | expr
keyargs ::=
    { name '=' expr ',' } ( name '=' expr | key '...' )
qualifier ::=
    { '.' name | '[' exprlist ']' | '{' exprlist '}' }
array ::=
    [ '('list2d ')' | '[' list2d ']' | '{' exprlist '}' | '(' generator ')' | '[' generator ']' | '{' generator '}' ]
list2d ::=
    exprlist { ';' exprlist } [ ';' ]
generator ::=
    expr ':' iter { ';' iter } [ '|' expr ]

```