PM –Parallel Models Programming Language

Version 0.3

Revision 00

Language Reference (incomplete)

Contents

1. Introduction 7
   - i. Background 7
   - ii. Inspiration 7
   - iii. Executing a PM program. 7
   - iv. Conventions used to define language syntax 7
2. Lexical Elements 8
3. Modular Structure 10
   - i. Modules 10
   - ii. Importing declarations from other modules 10
4. Types and values 12
   - i. Types and values 12
   - ii. Type conformance 12
   - iii. Universal Type 12
   - iv. Null type 12
   - v. Integer types 12
   - vi. Real Types 13
   - vii. Complex types 13
   - viii. Numeric type balancing 13
   - ix. General numerical types 14
   - x. Boolean type 14
   - xi. String type 14
   - xii. Proc and name types 14
   - xiii. Structures and records 15
   - xiv. Tuples 17
   - xv. Optional types 17

5. User defined types 19

i. Type declarations 19

ii. Parameterised type declarations 19

iii. Open type declarations 19

iv. Recursive type declarations 20

6. Procedures 21

i. Procedure definitions 21

ii. Procedure calls 21

iii. Variable-length argument lists 21

iv. Keyword arguments 22

v. Procedure signatures 23

vi. Matching procedure calls to procedure definitions 24

vii. Procedure specialisation 24

viii. Type constraints between different arguments to a procedure 24

7. Objects 25

i. Variables and constants 25

ii. Concrete types 25

iii. Changing values in a variable – assignment and reference arguments 26

iv. Object resource management 26

v. Each procedures 27

8. Expressions 28

i. Operator expressions 28

ii. Sub-expressions 28

iii. Parameter declarations 29

9. Sequential control statements 30

i. If statement 30

ii. Select statement 30

iii. Repeat statement 30

iv. While statement 30

v. For each statement 31
vi. Check and debug statements 32
10. Polymorphism and dynamic dispatch 33
i. Polymorphic types 33
ii. Dynamic dispatch using the any statement 33
iii. Object-oriented programming in PM 34
11. Sequences 35
i. Ranges 35
ii. Sequences 35
iii. Cyclic sequences 35
iv. Open ranges and sequences 36
v. Generic sequence types 36
12. Domains 37
i. Domains 37
ii. Grids 37
iii. Matrix and vector domains 38
iv. Domain of a sequence or tuple of sequences 38
v. Domain shapes 38
13. Arrays, Vectors and Matrices 39
i. Dimensions 39
ii. Arrays 39
iii. Subscripts and slices 40
iv. Vectors and matrices 41
14. Distributions 42
i. Overview 42
ii. Tiles 42
iii. Block distribution 42
iv. Fixed block distribution 42
v. Cyclic distribution 42

vi. Block cyclic distribution 42
vii. The distribute procedure. 43
15. Parallel execution 44
i. The for statement 44
ii. Returning values from a for statement 44
iii. Parallel Search 45
iv. Task parallelism 46
16. Processor allocation 47
i. Tasks, processors and processor ownership 47
ii. Concurrent clause 48
iii. The using clause 48
iv. Topology 48
v. Partitioning 48
vi. Work-sharing 49
17. Communicating operations 50
i. Overview 50
ii. Invariant values 50
iii. Channel variables 50
iv. Channel subscript: %{} 51
v. Neighbourhood subscript: @{} 51
vi. Communicating array subscript {} 52
18. Distributed Arrays 53
i. Overview 53
ii. Distributed dimensions 53
iii. Parallel scope 53
iv. Distributed array creation and access 53
v. Distributed arrays in for and find statements 54
19. Communicating procedures 55
i. Overview 55

ii. Implicit arguments to communicating procedures 56
iii. Reduction procedures 56
iv. Local procedures 56
v. Global procedures 56
20. Structured parallel communication 58
vi. Overview 58
vii. The sync statement. 58
viii. Synchronisation matching rules 58
ix. The if invar statement 59
21. Syntax 60
22. Intrinsic procedures 64
i. Arithmeric operations 64
ii. Numerical comparisons 64
iii. Numerical conversions 64
iv. General comparisons 64
v. Logical operations 65
vi. String operations 65
vii. Array operations 65
viii. Type comparison 65
ix. Ranges and sequences 66
x. Domains 66
xi. Distributions 67
xii. Processor grouping 67
xiii. Communicating procedures 67

1. Introduction

i. 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 introduces a new paradigm – communicating operations – which combines the ability of PGAS languages to use straightforward subscripting to access remote data with a robust, structured approach to synchronisation.

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 model
- The language should concentrate on parallel structures required for numerical computation
- 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.

ii. Inspiration

PM draws inspiration from many sources. Notable influences include: ZPL (http://research.cs.washington.edu/zpl/overview/overview.html), Parasail (parasail-lang.org), Go (golang.org) , NESL (http://www.cs.cmu.edu/~scandal/nesl.html) and llc (Reyes *et al*.,2009, 16th European PVM/MPI Users Group Meeting). Acknowledgement must also be made to two recent languages tackling similar problems: Chapel (chapel.cray.com) and Julia (julialang.org); ideas have been adapted from both.

iii. Executing a PM program.

A system for executing PM programs could be implemented using a variety of means (interpreters, direct compilation, just-in-time (JIT) compilation, source-to-source compilation, etc.) In terms of the language specification the following requirements are placed on any development system (which may itself consist of a suite of development tools).

- PM semantics require a global program analysis to be conducted prior to program execution
- At least one program execution option must gracefully halt program execution with an appropriate error message if an error condition occurs.
- At least one program execution option must be optimised for performance (in terms of an appropriate balance of execution speed, resources use and communications latency.) In this option the behaviour of the program if it enters and error state may be undefined.

iv. Conventions used to define language syntax

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

name ::= *list*	Define a non-terminal element in terms of other elements
name	Non terminal element
dim	Keyword
'>='	Character combination
[*elements*]	Elements are optional – may appear zero or one times
{ *elements* }	Elements may appear zero, one or more times
el1 | *el2*	Exactly one of the listed element sequences must be present
(el1 | el2)	Brackets may be used to group selections that are not enclosed by {} or []

2. Lexical Elements

A PM program is defined using a set of ***modules***. Each module defined using a text file (or equivalent). PM module names are associated with file names (or equivalent) in an implementation specific manner. A valid PM module file will have lines of no more than 1000 characters.

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 contexts where a semicolon **';'** is expected/allowed by the syntax, 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.

name ::= ['_']*letter*{'_'|*letter*|*digit*}

Names that start with '_' 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:

also	**and**	**arg**	**argc**	**by**	**check**
conc	**const**	**debug**	**default**	**dim**	**do**
else	**elseif**	**endany**	**enddebug**	**enddo**	**endfor**
endif	**endproc**	**endselect**	**endtype**	**endwhile**	**find**
for	**global**	**if**	**in**	**include**	**includes**
invar	**false**	**is**	**key**	**not**	**null**
or	**otherwise**	**over**	**param**	**present**	**proc**
rec	**reduce**	**render**	**repeat**	**return**	**require**
result	**select**	**struct**	**then**	**true**	**type**
until	**using**	**when**	**where**	**while**	**with**

Numeric constants take the following forms***:***

```
number              ::=   ( integer_constant | real_constant | imaginary_constant ) [ bits ]
integer_constant    ::=   { digit }  [ ( 'r' | 'R' ) { ( digit | letter ) }  ] [ 'l' | 'll' | 'L'  |'LL'  ]
real_constant       ::=   { digit }  [ '.' digit { digit } ]  [ ( 'e' |'d' | 'E' | 'D' )  [ ['+' | '-' ]  digit { digit } ] ]  ]
imaginary_constant  ::=   [ integer_constant | real_constant] ( 'i' | 'j' | 'I' | 'J')
bits                ::=   '_' { digit }
```

Details for interpreting numeric constants are given in the section on numeric types.

String constants are enclosed by double quotes '"' and may include any character other than '"'. A '"' character in a string is represented using '""' without any white space between the '"' characters. String constants may not span more than one line.

```
"Hello World"
```

3. Modular Structure

i. Modules

A PM program consists of one or more ***modules***. A module defines a collection of ***procedures***, ***types*** and ***parameters*** (module level constants). It also implicitly defines ***structure and record tags***. A ***program module*** additionally contains executable statements. A ***library module*** may optionally contain a **debug** statement containing module testing code but may not contain any other executable statements. Module names are constructed from a sequence of standard PM names 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 }
```

All names used by the PM language may optionally start with an underscore character. Such names are local to the module in which they are used and do not match or conflict with names in any other module.

Type, procedure and parameter declarations follow any **include** statements in the module. Types, procedures and parameters occupy separate name spaces – they may have the same name as each other.
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 used by other modules. Library modules may be used by other modules. They may not contain executable statements after the declarations except for an optional single **debug** statement that should contain module testing code.

ii. Importing declarations from other modules

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

Modules may import declarations from library modules using an **include** declaration. The default form of this declaration imports all parameters, types, procedures and structure tags defined in the named module into the current module. However, the importing process does not import definitions indirectly accessed through **include** statements in the imported module. An **include** statement must not result in conflicting definitions. Name clashes may be avoided by using a qualifying clause in the **include** statement. This lists specific elements to import and optionally renames them:

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

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

Only items listed in the qualifying clause are imported from the given module. In this context, the => symbol causes definitions accessed via a given name in the imported module to be made available using a different name in the current module. If an item in the qualifying clause is not specified as **type**, **param**, **proc** or **struct** then it is assumed to be a procedure (**proc**).

4. Types and values

i. Types and values

The PM type system provides flexibility similar to some dynamically typed languages while being designed for static analysis. Run time type inference only occurs within a specific control structure: the **any** statement. Implementing the PM type system requires static type analysis between and across modules.

PM programs operate on ***values*** which are ***grouped*** using types.

- A ***value*** is a particular representation of a piece of information, such as a 16-bit binary encoding of the number 2.
- A ***type*** consists of a (possibly infinite) set of values and/or other types and is defined using a ***type constraint expression*.**

ii. Type conformance

C*onformance* relationships are defined between types and between values and types:

- A value V conforms to an type T if V∈T or for some type U, V∈U and U conforms to T
- Type T conforms to type U if T∈U or for some type V, T∈V and V conforms to U

Both of these definitions are applied recursively if required.

iii. Universal Type

The universal type, which includes all other types, is denoted using the keyword **any**. Explicit use of **any** to denote the universal type is usually not required; the absence of a type constraint serves a similar purpose in most cases where a type expression is expected, including the components of a composite type constraint expression. All types conform to **any** (including **any** itself). All values conform to **any**.

iv. Null type

The **null** type contains the single value **null**. No other type conforms to **null**.

v. Integer 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 a strict bit size should use inquiry functions to check that a given type meets the expected requirements. Each integer type defines a set of distinct values that conform to it. No other type conforms to an integer type.

int	*short integer*	*System defined standard integer*	
long	*long integer*	*Integer capable of counting elements in largest possible array*	
long_long	*Long long integer*	*Integer capable of holding offsets in the largest possible file*	
int8	*8-bit integer*	*OR smallest integer holding -127..+127*	
int16	*16-bit integer*	*OR smallest integer holding -32767..+32767*	
int32	*32-bit integer*	*OR smallest integer holding -2147483647..+ 2147483647*	*OR same as* **int**
int64	*64-bit integer*	*OR smallest integer holding -9,223,372,036,854,775,807.. +9,223,372,036,854,775,807*	*OR same as* **int32**
int128	*128-bit integer*	*OR smallest integer holding -170,141,183,460,469,231,731,687,303,715,884,105,727.. +170,141,183,460,469,231,731,687,303,715,884,105,727*	*OR same as* **int64**

Integer constants are usually designated using whole decimal number, but may also 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**' representing 10 to 36 respectfully.

```
123        2r101011101   16rdeadbeef
```

By default integer constants yield values conforming to **int**. Long integer constants should append an **l** or **L** to the value. Long long integer constants should append **ll** or **LL** (note only **L** or **LL** may be used after a constant using **r**). For other integer types, append an underscore and the corresponding (requested) number of bits:

```
889874324897284746l                           ! long
2r1010101010101010101010101010010101L         ! long
12345678904567889874324897284746ll            ! long long
2r10101010101010101010101010101001010101101LL ! long long
255_8                                         ! int8
2r1010101010101010101010101010_32             ! int32
```

vi. Real Types

PM defines a set of real types. Each is associated with a distinct set of floating point values that conform to it. No other type conforms to a real type.

Real	*Standard (implementation defined) floating point value*		
Double	*Standard (implementation 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 and underscore and the assumed number of bits:

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

vii. Complex types

Complex types are defined as follows. No other type conforms to a complex type.

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*

Imaginary constants terminate with a letter '**i**' or '**j**'.

```
3.0i        ! cpx
1.0i_64     ! cpx64
-2.2-30di   ! double_cpx
```

viii. Numeric type balancing

Values of different numeric types may appear together in numeric expressions. The rules more mixed types are:

The most general type is selected for the result(s) and the least general value converted to that type. Generality is defined as follows (least..most):

int	**long**	**int8**	**int16**	**int32**	**int64**	**int128**	**real**	**double**
real32	**real64**	**real128**	**cpx**	**double_cpx**	**cpx64**	**cpx128**	**cpx256**	

ix. General numerical types

More general categories of numerical values are defined using the following types:

any_int	**int, long, int8, int16, int32, int128**	
any_real	**real, double, real32, real64, real128**	
any_cpx is	**cpx, double_cpx, cpx64, cpx128, cpx256**	
int_num	**any_int**	*
real_num	**any_int, any_real**	*
cpx_num	**any_int, any_real, any_cpx**	*
num	**cpx_num**	*
std_int	**int, long**	
std_real	**real,double**	
std_cpx	**cpx, double_cpx**	
std_num	**std_int, std_real, std_cpx**	

Types marked with * may be extended using the **also includes** mechanism (see User Defined Types.)

x. Boolean type

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

xi. String type

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"
```

xii. Proc and name types

A **proc** type contains a single value that is a valid PM procedure name (or operator symbol). Both procedure types and procedure constants are formed by preceding a name by **proc.**:

> **proc** '**{**' *procname* '**}**'

```
proc{cell_model}
proc{+}
```

The **proc{}** syntax may be omitted for procedure names which are PM names (not operator symbols or reserved words) that are not the same as the names of local variables or constants and not the same as any visible parameter definition.

All procedure types conform to the type **proc**.

A name type contains a single value. Both a name type and its corresponding name value is designated by:

> '**$**' *name*
> '**$**' **true**
> '**$**' **false**

All name values and name types conform to the type **name**.

It is possible to create variables and constants with a **proc** or **name** type. However, a variable holding a proc or name value can only be assigned that same value (this seemingly useless assignment operation is permitted to enable generic code containing assignments to be able to be applied to name values.) Proc or name values do not usually require any runtime storage. They can provide a mechanism for embellishing the type of a structure or record value, providing information that can influence procedure selection.

xiii. Structures and records

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

A structure or record type is defined using the following type constraint expression:

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

For example:

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

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

[**struct** | **rec**] [*name*] '**{**' [**include**] *name* '**=**'*exp* { '**;**' [**include**] *name* '**=**'*exp* } '**}**'

For example:

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

A structure or record value associates each component with a name. Component names are local to the structure or record.

A structure or record value may be associated with a ***tag***. This is a standard PM name and is global to the PM program (unless starting with an underscore, in which case it is local to the module). Within a given module, a structure or record value or type constraint with a given tag must be associated with the same component names (not necessarily listed in the same order). This ***tag matching rule*** applies irrespective of the kind of aggregate. A structure and a record with the same tag must have the same component names. The matching rule applies to a limited extent between modules. If module *A* includes module *B* using an unqualified **include** declaration then the tag matching rule applies between all structure and record values and type constraints defined in the two modules. If a qualified **include** is used then the matching rule only applies to tags explicitly listed in a **struct** entry in the qualifying clause (note that while the **struct** keyword is used in the qualification, tag matching between **struct** and **rec** tags still applies).

A structure or record value conforms to a given structure or record type constraint 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 name tagging the value matches the corresponding name tag given in the **struct**/**rec** type, or both are absent
3. The same component names are present (not necessarily in the same order)
4. The same component names (if any) are associated with the **include** keyword.
5. The value for each named element in the structure or record conforms to the type constraint associated with the same name in the type constraint expression (if such a constraint is present).

A structure or record type *T* conforms to a structure or record type *U* if:

1. Either both *T* and *U* are record types or both are structure types.
2. The name tag for *T* is the same as the name tag for *U*, or both are absent.
3. *T* and *U* have the same number of components with the same component names (not necessarily in the same order).
4. The same component names (if any) are associated with the **include** keyword.
5. The type constraint associated with each named element in *T* must conform to the type constraint associated with the same element name in *U*.

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)
```

In order to maintain the no-modification rule, a '**.**' operation by not be applied to a record on the left hand side of an assignment or in the expression defining a reference parameter in a procedure call ('**&**')

If an element in a structure value was defined with the **include** keyword present and if the value of that element is a structure or record, then the elements of that element may be referenced as if they were elements of the parent value.

```
x:=rec {include a=rec{p=1,use q=2},b=3}
print(x.a.p)     ! Outputs 1
print(x.p)       ! Also outputs 1 – accesses the same element
```

The process works recursively:

```
x:=rec {include a=rec{p=1,include q=struct{p=4,m=5}},b=3}
print(x.p.q.m)   ! Outputs 5
print(x.m)       ! Also outputs 5 – accesses the same element
```

This form of access to ***embedded elements*** is only applies if the name of the embedded element is not the same as the name of an element in the parent value – parent elements always shadow embedded elements. In addition, if embedded element access could ambiguously refer to more than one element then it is illegal.

```
x:=rec {
  include a=rec{
    include p=struct{q=4,r=5},
    q=4
  },
  include b=rec{
    r=4
  }
}
print(x.q)       ! This validly refers to x.a.q which shadows x.a.p.q
print(x.r)       ! Error – could refer to x.a.p.r or x.b.r
```

The use of embedded element access is primary designed to facilitate code reuse. A generic procedure operating on the elements of structure or record *S* will also be applicable to a value **rec {include s=***S***, … }**.

xiv. Tuples

A tuple is similar to a record. Tuples have one to seven elements named **d1** .. **d7** that may be accessed using the same dot notation as for a record component. A tuple type is defined as:

tuple '**{** [*type*] { '**,**' [*type*] } '**}**'

Tuple types with a given number of components of any type are defined as:

tuple1d	**tuple{}**
tuple2d	**tuple{,}**
tuple3d	**tuple{,,}**
tuple4d	**tuple{,,,}**
tuple5d	**tuple{,,,,}**
tuple6d	**tuple{,,,,,}**
tuple7d	**tuple{,,,,,,}**

A tuple value is defined using the tuple constructor:

'**[**' [*expr*] { '**,**' [*expr*] } '**]**'

If an element of a tuple constructor is missing, it is given the value **null**. A tuple constructor is syntactic sugar for a call to the **tuple** intrinsic procedure.

A tuple value conforms to a tuple type if the type has the same number of elements as the value and each element, in order, conforms to the corresponding element in the tuple type.

A tuple type *V* conforms to a tuple type *U* if *V* and *U* have the same number of elements and each element of *V*, in order, conforms to the corresponding element of *U*.

xv. Optional types

A value conforming to an optional type has an internal representation that is capable of representing any value of a given type, or a null value. Note that the representation of an optional value will not generally use less space for a null value than for a non-null value. The type constraint expression for an optional type has the following form:

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

Values of an optional type are created using one the following procedure calls

```
y:= opt(3.3)
      ! Creates an optional real value holding the value 3.3
z:= null(2)
      ! Creates an optional integer value holding the value null
x:= opt(2.0,a>b)
      ! Creates an optional real value holding 2.0 if a>b or null
      ! otherwise
```

The value contained within an optional value may be obtained using the **val()** procedure or the '**|**' operator. Both return the value contained within the optional value if it is not null. The former creates an error condition if the optional value is null. The latter returns the value to the right of the operator when the optional value is **null**. The **isnull** function tests if a value of an optional type contains **null**.

```
y:=opt(2.0)
z:=null(2.0)
w:=val(y)        ! Sets w to 2.0
x:=val(z)        ! Error!
a:=y|1.0         ! Sets a to 2.0
b:=z|1.0         ! Sets b to 1.0 (value to the right of | is the default)
print(isnull(y)) ! prints false
print(isnull(x)) ! prints true
u:=1.0
```

An optional value created using **opt(**x**)** (or equivalent) conforms to type constraint **opt{***T***}** if x conforms to *T*.

The type constraint expression **opt{***T***}** conforms to **opt{***U***}** if *T* conforms to *U*.

5. User defined types

i. *Type declarations*

Type declarations associate a name with a given set of type constraint expressions, defining a new type as the set of the listed types:

typedecl::=
    type *name* ['**{**' *typeparams* '**}**'] [**in** *namelist*] (**is** *typelist* | **includes** *typelist* | **also includes** *typelist*)
typeparams::=
    name ['**:**' *type*] { '**,**' *name* ['**:**' *type*] }

The simplest type declaration simply associates a name with a set of types:

```
type x is int, struct{x:int,y:int}
```

ii. *Parameterised type declarations*

A ***parameterised type*** declaration defines a template that may be parameterised by one of more ***type arguments*** in order to generate an ***actual type*** that may be used in a constraint expression. An actual type is derived from a type template by providing the correct number of type constraint expressions as type arguments.

```
type point{t:num} is struct{x:t, y:t}
type integer_point is point{int}
type num_point is point{}
```

Type arguments used to create an actual type must conform, parameter by parameter, to the corresponding parameter type constraints, or be missing. Missing type arguments are assumed to be equal to the corresponding type parameter constraint or to **any** if no such constraint is present.

Type templates are characterised by their name and number of parameters, but not their respective type parameter constraints. Thus **R{t}**, **T{t}** and **T{t,u}** each refer to different type templates while **T{t,u}**, **T{r,s}** and **T{,}** refer to the same type template . Two type template definitions may not both refer to the same template name if they also have the same number of parameters.

iii. *Open type declarations*

An ***open type declaration***, using the **includes** syntax provides a potentially incomplete definition for a type which may be added to by other definitions. The type definition may be augmented using an **also includes** type definition or by using by declaring another expression to be **in** that type:

```
type a includes int, real
type a also includes string
type b in a is cpx    ! Type a is int, real, string, b
```

Augmenting type declarations do not have to be in the same module as the original **includes** declaration and do not have to precede the **includes** declaration in the module or program text.

Open type declarations may also be parameterised:

```
type point{T:num} includes rec point{ x:T,y:T }
type point{T} also includes rec point{ x:T,y:T,z:T}
```

If an **also includes** type definition is parameterised, then the type constraints associated with its parameters must either be absent or must conform, parameter by parameter, with the type constraints in the **includes** definition to which it refers. If type constraints are present, the **also includes** definition only augments the type template when the type arguments in the actual type expression conform to the types parameters in the **also includes** definition. For example:

```
 type point{T:int16} also includes rec point{ combined_index: int32 }

      ! point{int) matches rec point{x:int,y:int}
      ! point{int16) matches both:
      !                            rec point{x:int,y:int} and
      !                            rec point{ combined_index:int32 }
```

iv. Recursive type declarations

A PM type may not conform to itself (and thus may not be a member of itself or a member of any type that conforms to it). However, type constraint expressions used to define a type *T*, or a type conforming to *T* may contain *T*. This enables the definition of recursive types.

```
type list is struct{head:int,tail:list},int
```

Note that while useful, directly recursive types scale poorly as each level of recursion introduces a new type. The above definition would therefore only be suitable for short lists.

PM type definitions may create infinite types:

```
type list is struct{head:int,tail:list}
```

Such declarations are legal since they may occur during program development (an implementation may decide to provide warnings). An infinite type is essentially a null type since no finite value will conform to it.

6. Procedures

i. *Procedure definitions*

A procedure defines an operation on a set of objects, which may change some of their stored values. Procedures may also return one of more values.

> **proc** *procname params* '**=**' *xexprlist* [**do** *statements* **endproc**]
> **proc** *procname params* [**check** *exprlist* [*whereclause*]] **do** *statements* [**result** '**=**'*xexprlist*] **endproc**
>
> *params*::=
> '**(**' [*pars* ['**,**' *keypars*] | *keypars*] '**)**'
>
> *pars*::=
> { *param* '**,**' } (*param* | **arg** '**...**')
>
> *param*::=
> ['**&**'] *name*['**:**' *type*]

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: real[]) do
     ......
     result=mean,std_dev
endproc
```

ii. *Procedure calls*

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.

```
u, 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 value. Thus the following are equivalent:

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

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

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 '**&**'.

iii. *Variable-length argument lists*

A procedure may accept a variable number of arguments:

```
proc set_numbers(&dest:int[],arg:int...) do  ....
```

It is also possible to pass optional arguments *en masse* to a procedure call (***a pass through*** 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)
```

iv. Keyword arguments

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

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.

```
procrun_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.

v. Procedure signatures

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

+ - * / ** == /= > >= and or // => dim = [] {} @{} @{}| .. ..._ _... by not null

Note the absence of **<** or **<=** from this list – these operators are defined with respect to > or >=.

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 constrains on the procedure's parameters
5. Which parameters are declared to be reference parameters
6. The number of channel variable parameters
7. Which parameters are required to be invariant

Items 6-7 are defined in the section on communicating procedures.

Given that procedure names may be changed when imported to another module, a given procedure definition may give arise to different signatures in different modules.

A procedure signature *P* is said to ***conform to*** procedure signature *Q* if:

1. The procedure names are the same.
2. Either
   a. Both P and Q have the same number of parameters.
   b. P accepts a variable number of arguments and has fewer parameters than Q
   c. Q accepts a variable number of arguments and has fewer parameters than P
   d. Both P and Q accept a variable number of arguments
3. 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*. In either case the corresponding parameter may be in the variable-length part of the argument list, in which case the type constraint for **arg...** applies.

Signatures *P* and *Q* are said to ***conflict*** if both *P* conforms to *Q* and *Q* conforms to *P*. Two conflicting signatures may not be present in the same module, whether through direct definition, importing declarations from other modules or any combination of the two.

Signature *P* is ***strictly more specific*** than signature *Q* if *P* conforms to *Q* and either:

1. Type constraints are not strictly equal to one another for at least one parameter position
2. *P* has a greater number of formal arguments declared **invar** than does *Q*.

Two signatures *P* and *Q* are said to be ***ambiguous*** if:

1. P is not strictly more specific than Q
2. Q is not strictly more specific than P
3. There exists a possible procedure call which conforms to both P and Q (the call does not have to be present in the program, just involve a theoretically possible combination of argument types).

Ambiguous signatures may not be present in the same module unless the ambiguity is ***resolved***. Resolution of an ambiguity between *P* and *Q* occurs if there exists a third signature *R* for which:

R is strictly more specific than *P*
R is strictly more specific than *Q*
Every possible procedure call that conforms to both *P* and *Q* also conforms to *R*.

vi. Matching procedure calls to procedure definitions

When encountering a procedure call, PM finds all procedures with signatures that ***conform to*** the call. A call conforms to 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 value supplied for each argument conforms to the type constraint for the corresponding parameter, or to the type constraint for **arg…** if the argument corresponds to the variable-length portion of the parameter list.
5. The number of channel variable parameters matches the number of channel variable arguments
6. Parameters constrained to be loop-invariant (**invar**) are associated with loop-invariant arguments

Items 5-6 are described in the section on communicating procedures. 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***, leading to an error.

Arguments passed using **arg…** pass-through form part of the signature of a procedure call – they are used in the matching process and do not necessarily have to correspond directly to the variable argument portion of the called procedures parameter list.

vii. Procedure specialisation

Procedure specialisation is an optimisation of procedure call selection. The PM system should construct a specialised implementation of the procedure body for the specific combination of concrete types associated with the argument values present in the procedure call. If this optimisation is not implemented, there should be no significant loss in runtime performance compared an implementation strategy that adopts this optimisation.

viii. Type constraints between different arguments to a procedure

Unlike some other languages adopting ‘multi-method’ approaches to procedure selection, PM does not provide a mechanism within procedure selection to specify that the type of one argument (or component thereof) should match the type of a second argument. The **same_type** function provides a similar functionality. This procedure returns **$false** its two arguments have different types and **$true** if the two arguments have the same type. A **check** statement will fail at compile time if given and expression which evaluated to **$false**. For those cases where the type matching should modify procedure selection, the **same_type** procedure may be used as follows.

```
proc _set_table(&table:table,value,same:$true) do
      _set_table_from_value(&table,value)
endproc
proc _set_table(&table:table,value,same:$false)
check "Attempt to set tables of different types":same_type(value,table) do
      _set_table_from_table(&table,value)
endproc
proc set_table(&table,value) do
      _set_table(&table,value,same_type(table.first_element,value))
endproc

set_table(&table1,0)             ! Fill with values
set_table(&table2,table1)        ! Copy values between tables
```

7. Objects

i. Variables and constants

PM stores values in ***named objects*** which may be ***variables*** or ***constants***. Variables may changes the value stored in them, constants may not. Objects are created and associated with names using a declaration statement.

definition::=
 vlhs '**=**' *expr*
 vlhs '**,**' *vlhs* { '**,**' *vlhs* } '**=**' *call*
 name ('**:=**' | '**::=**') *expr*
 name '**,**' *name* { '**,**' *name* } ('**:=**' | '**::=**') *call*
vlhs::=
 '**_**' | **var** *name* ['**:**' *type*]| **var** '**{**' *namelist* '**}**' ['**:**' *type*]| **const** *name* ['**:**' *type*]

For example

```
x:= 4
pi::=3.14179
var x : int = 3
const message = "Hello World"
var a,const b = f(0)        ! f() must return two values
var {x,y,z},const y = f(0)  ! x,y,z set to first value returned by f
                            ! y is set to second value returned by f
```

A declaration consists of either a single **var** or **const** clause which is associated with an initial value using the = operator. Multiple left hand side clauses may be set equal to multiple values returned from a procedure call. A compound **var** clause, listing a set of names in braces, creates multiple variables set to the same initial value.If all of the left-hand-side clauses are simple **var** clauses, then the **:=** operator may be used and the **var** keywords omitted. If all of the left hand side clauses are **const**, then the **::=** operator may be used in the same manner.

A name is defined until the end of the block of statements in which it is defined. Variable or constant names may not shadow variable or constant names defined in enclosing blocks or clash with the names of procedure parameters.

ii. Concrete types

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

1. If the value, *x*, is a ***resource template*** then the concrete type is the type of the value returned by resource creation.
2. If the 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 proc** then that type is the concrete type of the object.
3. If the value is a structure or record, then the concrete type is a structure or record type with component types given by the concrete types of the corresponding components, determined by applying these rules recursively.
4. If the value is optional created using **opt(***x***)**, or equivalent, then the concrete type is **opt{***T***}** where *T* is the concrete type of *x*.
5. If the value is a polymorphic value created using **<** *T* **>***value* then the concrete type is **<***T***>**
6. If the value is an array, then the concrete type is an array type *D***[***E***]** with domain (*D*) and element (*E*) types equal to the concrete types of the domain and an arbitrary element of the array value.

iii. Changing values in a variable – assignment and reference arguments

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. The new value stored in the variable must conform to the concrete type of the value used to create the variable.

```
assignment::=
        lhs '=' expr
        lhs ',' lhs { ',' lhs } '=' call
lhs::=
        ref | vlhs | '_'
ref::=
        name qualifier
```

```
proc double(&x)=x*2
x := 4       ! Value stored in x is 4
x = x + 1    ! Value changed to 5
double(&x)   ! Value changed to 10
```

Some procedures return more than one value. In this case, multiple left hand sides are permitted. An underscore may be used as a place filler, allowing a given returned value to be ignored. For example:

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

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
```

It is possible to mix assignment and object creation operations:

```
 a:=0
var x,a = f(x)
```

iv. Object resource management

In addition to holding one of a defined set of values, objects may also be associated with resources. Such resources may include, for example, the memory used to hold an array or a reference to an external file. These resources are obtained and attached to the object at the point of object creation. They are detached from the object and released when the object goes out of scope. Some resources may be moved between objects using intrinsic functions. However, a given resource may not be attached to more than one object.

As noted above, resource allocation occurs at object creation. Certain values are interpreted by the object creation process (invoked by **:=**, **::=**, **var =** or **const =**) as providing templates for resource allocation rather than a simple initial value. An example of this is the **dim** value. A **dim** value is returned by the **dim** operator and provides a template for array creation. The following statement:

```
array:= 1 dim grid(1..3)
```

first creates a **dim** value containing the array domain and the value to be used to initialise the elements. The **:=** operation creates an array object from the template provided by the **dim** value, obtaining the memory required to store the array. This memory is relinquished when the `array` variable goes out of scope.

v. *Each procedures*

An **each** procedure is a specialised form of procedure declaration that enables specialised treatment to be given to particular components of an object, irrespective of the form of the larger object within which those components are embedded. An **each** clause lists one or more parameters. If the argument corresponding to the first parameter listed in **each** is a **struct** or **rec** value, then the arguments associated with the remaining each parameters are required to be structure or record values with the same tags (if present) and component names. In this case, the procedure recalls its same signature, once for each component of its **each** argument(s). It the argument associated with the first **each** parameter is not a **struct** or **rec** value, then the normal procedure body is executed.

each procedures are typically used in combination with other procedure declarations with the same signature that define specialised behaviour. For example, the following combination creates a procedure that will convert all **real** components of a value to **double**.

```
proc double_components(x) each(x) = x
proc double_components(x:real)=double(x)
a:=double_components(rec{u=1.0,rec {v=2.0}})
     ! Same as a:= rec{u=1.0d,rec{v=1.0d}}
a:=double_components(rec{u=true})
     ! Procedure has no effect (since default behaviour is =x)
```

8. Expressions

i. Operator 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 or communicating procedure calls.

Operation	Priority	Equivalent call	Description
`x[a,b … ]`	0	`proc [] (x,a,b…)`	Array subscript
`x{a,b, … }`	0	`proc {} (x,a,b…)`	Communicating array subscript
`x%{a,b,… }`	0	`proc {}%x(a,b,…)`	Channel subscript
`x@{a,b,… }`	0	`proc @{}%x(a,b,…)`	Channel neighbourhood subscript
`x | y`	2	`proc | (x,y)`	x default value y
`x**y`	3	`proc ** (x,y)`	Power
`x*y`	4	`proc * (x,y)`	Multiplication (including matrix multiplication)
`x/y`	4	`proc / (x,y)`	Division
`-x`	5	`proc – (x)`	Minus
`x mod y`	6	`proc mod(x,y)`	Modulo
`x + y`	7	`proc + (x,y)`	Addition
`x – y`	7	`proc – (x,y)`	Subtraction
`x .. y`	8	`proc .. (x,y)`	Range creation
`... y`	8	`proc _... (x,y)`	Partial range creation
`x ...`	8	`proc ..._ (x,y)`	Partial range creation
`by x`	8	`proc by (x)`	Striding value creation
`x by y`	9	`proc by (x,y)`	Sequence creation
`x dim y`	10	`proc dim(x,y)`	Array creation
`x == y`	12	`proc == (x,y)`	Equals
`x /= y`	12	`proc /= (x,y)`	Not equal
`x > y`	12	`proc > (x,y)`	Greater than
`x >= y`	12	`proc >= (x,y)`	Greater than or equal
`x < y`	12	`proc > (y,x)`	Less than
`x <= y`	12	`proc >= (y,x)`	Less than or equal
`x in y`	12	`proc in (x,y)`	Membership
`x includes y`	12	`proc includes(x,y)`	Superset or equal
`not x`	13	`proc not(x)`	Logical not
`x and y`	14	`proc and (x,y)`	Logical and
`x or y`	15	`proc or (x,y)`	Logical or
`x # y`	16	`proc #(x,y)`	Format as string
`x // y`	17	`proc // (x,y)`	String concatenation
`x => y || z`	18	`proc=>(x,y,z)`	If x is true then y else z

ii. Sub-expressions

xexpr::=
    expr subexp
subexpr::=
    [**check** *exprlist*] *whereclause*
whereclause::=
    { **where** *constdefs* }

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)
```

It is also possible to include a check clause after an expression. This raises an error if the associated logical expression does not yield a **true** value:

```
x= my_sqrt_fn(x) check x**2==old_x where old_x=x
```

A **check** clause may provide as associated error message (a constant string):

```
y = f(x) check "f returned -ve" : y>=0
```

Check expressions may optionally not be compiled or executed.

iii. Parameter declarations

A **param** declaration 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. There is no explicit restriction on the procedures employed within a **param** declaration expression – they may be interpreted as zero-parameter procedures with their own separate name space. Parameter names may be shadowed by variable and constant names. They themselves will shadow procedure names used as constants of type **proc**.

9. Sequential control statements

i. If statement

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

The **if** statement conditionally executes a list of statements:

```
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
```

ii. Select statement

select [*xexpr*] **when** *xexprlist* **do** *statements* {**when** *xexprlist* **do** *statements* } [**otherwise** *statements*] **endselect**

The **select** statement tests an expression against lists of values and executes the first statement list to be associated with a value matching the expression.

```
select digit
    when '1','3','5','7','9' do print("Odd")
    when '2','4','6','8' do print("Even")
    otherwise print("Not a digit!")
endselect
```

If the test expression is missing it is assumed to be equal to **true**.

iii. Repeat statement

repeat *statements* **until** *xexpr*

The **repeat** statement sequentially repeats a list of statements until an expression yields a true value (test at the end).

```
repeat
    x = x/2
    y = y+1
until x == 0
```

iv. While statement

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

The **while** statement executes a statement zero or more time while a given expression yields a true value (test at the start)

```
nbits=0
while x>0 do
    x = x/2
    nbits = nbits + 1
endwhile
```

In each of these cases, conditional expressions must return a **bool** value. An error will result if they do not.

v. For each statement

for each *iter subexpr* [**until** *xexpr* | **while** *xexpr*] **do** *statements* **endfor**

The **for each** statement executes its body for each element in order in a given ***iterable value*** (an array, domain, matrix or vector). Iteration order is defined by the domain of the value.

```
for each i in grid(1..2,3..4) do
       print(i.d1+10*i.d2)
endfor
```

will output `31 32 41` and `42` in that order.

It is possible to iterate over more than one iterable value, providing they have conforming domains. Iteration order is determined by the domain of the first (leftmost) iterable expression.

```
for each i,j in dom(a),a do
      print(i//"="//j)
endfor
```

If one of the iteration variables is assigned to, this will change the value for the corresponding element in the value being iterated over (assuming such assignment is possible – an error occurs if not).

```
for each i,j in dom(a),a do
      j=i**2
      ! Assigning to i would generate an error
endfor
```

There are two mechanism to prematurely terminate a **for each** loop. A **while** clause tests for termination at the start of the loop body while the **until** clause tests at the end of the loop body. Despite is textual position, the **until** clause has access to all variables defined in the loop body.

```
for each i in 1..10 while i**2<10 do
      print(i)
endfor

for each i in 1..10 until i_sq>8 do
      print(i)
      i_sq:=i**2
endfor

! Both loops print out 1 2 3
```

vi. Check and debug statements

Two statements allow for debugging and may optionally not be compiled and executed. The **check** statement checks if an expression yields a **false** or **$false** value and raises an error if this is the case (in the latter case, during program analysis. The **debug** statement executes a block of code only if debugging is enabled.

check [*string* ':'] *xexpr*
debug *statements* **enddebug**

For example:

```
check value /= null

check same_type(a,b)  ! Compile-time error if types do not match

debug print("Entering main loop") enddebug
```

The **check** statement can be supplied with a literal string which will form part of the error report if the check condition fails.

```
check "x out of range": 0<=x and x<=max_thresh
```

10. Polymorphism and dynamic dispatch

i. Polymorphic types

A ***polymorphic*** value has an internal representation that is capable of encoding any value conforming to a given type constraint.

A value of a given polymorphic type is created using a generating expression:

'**<**' *type* '**>**' *subterm*

A variable initialised to a polymorphic value may contain any value conforming to the given type constraint. Note that it is not permissible for **any** to conform to the given type constraint – it is thus not possible to have a completely generic polymorphic value.

The value contained within a polymorphic value may be obtained or changed using the binary ''' operator. This yields the value contained in the polymorphic value on its left providing that value has the same type as the value given on the right, otherwise an error condition arises. It is also possible to use the binary '**|**' operator. This returns the value contained in the polymorphic value on its left providing it has the same type as the value given on the right. Otherwise the value on the right is returned.

```
type int_or_string is int,string
a:= <int_or_string> 1
a= <int_or_string> "Hello"
print(a'"")           ! Outputs "Hello"
a'""="Bye"
print(a'"")           ! Outputs "Bye"
print(a|"")           ! Outputs "Bye"
print(a|2)            ! Outputs 2
```

A polymorphic type constraint expression has the following form:

'**<**'[*type*]'**>**'

A polymorphic value conforms to type constraint **<***U***>** if it was created using expression <*T*>*x* and *T* conforms to *U*.

A polymorphic type **<***T***>** conforms to **<***U***>** if *T* conforms to *U*.

ii. Dynamic dispatch using the any statement

In PM, dynamic (runtime) dispatch is achieved using a control structure (the **any** statement) rather than being associated with each subroutine call as would be the case in some other languages. The **any** statement has the following form:

any *name* ['**=**' *xexpr*] [**do** *statements*] [**return** *definitions*] **endany**

An **any** statement takes a named polymorphic object, extracts the object it contains and associates that extracted object with the original name for the scope of the statement. It then runs a body of statements, possibly returning values to the enclosing scope through variable or constant definitions in an optional **return** clause. If a variable or constant is returned, it must have the same type irrespective of the type of the polymorphic component value.

```
a:=<num> 3
any a do
  a=a+a
return
  var square=d*d where d=double(d)
      ! If this was just a*a then type would differ
      ! depending on the type of a – an error
endany
! At this point a==<num> 6 and a new variable square==9.0d
```

It is possible to specify an expression, rather than a polymorphic object (in which case the given name must not clash with existing constants, variables or parameters):

```
m:= rec { … , counter=<any_int>0 }
any x = m.counter return const count=long(x) endany
```

If the expression is a direct reference to an object component, then the given name may be used to update that component.

```
any x = m.counter do x=x+1 endany
```

It should normally be possible to for a PM system implement an **any** statement with no more execution overhead than a **select** statement – the body of the statement being resolved for each possible type that could be extracted from the polymorphic value.

iii. Object-oriented programming in PM

By combining the **any** statement with structure/record embedding, it is possible to implement the features (including type and code inheritance hierarchies) of dynamically dispatched object-oriented languages. However, run-time dispatch will only be used when necessarily and will be explicitly requested by the programmer. For example:

```
type shape includes rec _shape{npoints:int, … }
type triangle in shape is rec _triangle{include shape, … }
type square in shape is rec _square{include shape, … }
proc make_shape(npoints)= rec _shape{npoints=npoints, … }
proc make_triangle(point1,point2,point3)=
           rec _triangle{include shape=make_shape(3),… }
proc make_square(point1,point2,point3,point4)=
           rec _square{include shape=make_shape(4),… }
proc draw(shape:triangle) do … endproc
proc draw(shape:square) do … endproc
…
shapes:=<shape>make_triangle([0,0],[0,0],[0,0]) dim grid(1..N)
…
total_points:=0
for shape in shapes do
   any shape do
      total_points=total_points+shape.npoints
      draw(shape)
   endany
endfor
```

11. Sequences

i. Ranges

Range types represent closed intervals. Range values are created using the '**..**', operator. For numerical ranges mixed type promotion rules apply. *E.g.*:

```
range1:= 0..5                  ! Contains 0, 1, 2, 3, 4, 5
range2:= 0.0..5                ! Contains { x: 0.0 ≤ x ≤ 5.0 }
```

The two bounds of a range value *r* may be obtained using the procedures **low(***r***)** and **high(***r***)**.

It the lower bound is greater than or equal to the upper bound then the range contains zero values (although it may form the basis of a non-empty descending sequence, as described below)

The following intrinsic range type are defined:

range_base includes any_real
range is range{}
range { t: range_base }

Integer ranges may be interpreted as integer sequences with a unit stride. The function **step(***r***)** returns a value of one and **size(***r***)** returns **max(high(***r***)-low(***r***),0)** if *r* is a range of integer type.

ii. Sequences

A sequence type represents a sequence of values. Sequence values are created by applying the **by** operator to a range value. The second argument to **by** gives the step between sequence elements. Numeric balancing will apply between the low and high values of the range and the step argument.

```
integer_sequence := 1..6 by 2         ! 1, 3, 5
reverse_sequence := 6..1 by -2        ! 6, 4, 2
empty_sequence := 1..6 by -1          ! No values
real_sequence := 3.2 .. 5.4 by 0.7    ! 3.2, 3.9, 4.6, 5.3
```

Sequences may define values in ascending or descending order. An ascending sequence starts at the low bound of the range and successively adds the step (which must be positive) while the resulting value remains less than or equal to the second bound of the range. A descending sequence starts at the high bound of the range and successively adds the step (which must be negative) while the resulting value is greater than or equal to the low bound of the range.

Whatever the direction of the sequence, **low(***s***)** and **high(***s***)** define the lowest and highest values (**low(***x***)==***a* and **high(***x***)==**b when *s*>0 and **low(***x***)==***b*, **high(***x***)==***a* when *s*<0 for *x***:=***a***..***b* **by** *s*). The first and last values of the sequence may be found using **first(***s***)** and **last(***s***)**. The step of a sequence may be found using **step(***s***)**. The number of elements in the sequence may be obtained using **size(***s***)**.

Sequence types are defined as:

seq is seq{}
seq { *t* **: range_base }**

iii. Cyclic sequences

A cyclic sequence is a sequence with cyclic connectivity. This means that when considering neighbouring values, the first element in the sequence directly follows the last element in the sequence. A cyclic sequence is constructed using the **cycle** procedure, which can take a sequence or an integer range as an argument.

```
x:= cycle(1..4)
y:= cycle(2..10 by 2)
```

Cyclic sequence conform to the type **cycle{*T*}** where *T* is the base type of the underlying sequence or range. They also conform to type **cycle**.

iv. Open ranges and sequences

An open range represents a set of values inclusively higher or lower than a given threshold. They are created using a prefix or suffix ... operator:

```
X:= ... 3              ! Contains { x: x ≤ 3.0 }
Y:= 2 ...              ! Contains { x: 2.0 ≤ x }
```

An open sequence is a strided open range:

```
xs:= ... 3 by 2
ys:= 2 ... by 3
```

It is also possible to have a strided infinite sequence created using a unary by operator

```
z:= by 2
```

Open ranges and sequences are most commonly used in subscript expressions.

v. Generic sequence types

The generic type **bd_seq{*T:range_base*}** contains **range{*T*}** and **seq{*T*}.** It conforms **bd_seq.**

The generic type **any_seq{*T:range_base*}** contains **cycle{*T*}** and **bd_seq{*T*}**. It conforms **any_seq.**

12. Domains

i. Domains

A domain represents a finite set of values which may be used to represent an iteration space, valid indices for an array, or valid indices for points in an iterable sequence or tuple of iterable sequences. PM domains are similar to domains in Chapel and regions in ZPL. A domain defines three things:

- A set of valid index values
- An order in which those values will be visited in a sequential iteration
- A connectivity between index values

Any value that may be iterated over (either sequentially or in parallel) will be associated with a domain. The domain of such a value *x* may be obtained using **dom(***x***).**

All domain values are iterable and form their own domain**: dom(dom(***x***))==dom(***x***)** always holds.

A domain value may or may not ***conform*** to another domain value. Conforming domains contain the same number of values and share a compatible structure (but not necessarily the same connectivity). Values with conforming domains may be processed in lockstep.

All domain values and types conform to the type **dom**.

ii. Grids

A grid is a domain value that defines an *N*-dimensional (1<*N*≤7) rectangular grid of values, each value consisting of a tuple of long integers. There are seven specific grid types of varying number of dimensions, and a generalised grid type encompassing all of these:

grid1d	**grid{grid_base, grid_base}**
grid2d	**grid{grid_base, grid_base}**
grid3d	**grid{grid_base, grid_base, grid_base }**
grid4d	**grid{grid_base, grid_base, grid_base , grid_base }**
grid5d	**grid{grid_base, grid_base, grid_base , grid_base , grid_base }**
grid6d	**grid{grid_base, grid_base, grid_base, grid_base, grid_base, grid_base}**
grid7d	**grid{grid_base, grid_base, grid_base , grid_base , grid_base , grid_base , grid_base}**
grid	**grid1d,grid2d, grid3d, grid4d, grid5d, grid6d, grid7d**

Each dimension of a grid may be defined using any value conforming to **grid_base** which includes:

- Any range of long integers
- Any sequence of long integers
- Any cyclic sequence of long integers

Grid values are created by calling the **grid** procedure:

```
model_grid := grid (0..num_cols, 0..num_rows)
```

For convenience, the **grid** procedure converts the base type of each of its arguments to **long** before constructing a **grid** value.

The value corresponding to each element of a grid value *G* may be obtained using: *G* **.d1**, *G* **.d2**, … , *G* **.d7**.

A grid *G* conforms to another grid *H* if they both have the same number of dimensions and the sequences associated with each dimension have the same number of elements:

conform(*G*:grid,*H*:grid)=(size(*G*.d1)==size(*H*.d1) and size(*G*.d2)==size(*H*.d2) and …)

The iteration order for a grid domain involves the leftmost index varying most rapidly. Thus the values in the domain **grid(1..2,10..20 by 10)** will be traversed in the following order:

[1l,10l]   [2l,10l]   [1l,20l]   [2l,20l]

iii. Matrix and vector domains

A vector domain is a specialised one dimensional domain corresponding to the rows of a mathematical vector. Vector domains conform to the type **vector{**dim**:bd_seq{long}}** and to **vector.**

A matrix domain is a specialised two dimensional domain corresponding to the rows and columns of a mathematical matrix. Matrix domains conform to the type **matrix{**dim1**:bd_seq{long}**, dim2**:bd_seq{long}}** and to **matrix.**

A vector domain is created using the **vector** procedure. A **matrix** domain is created using the matrix procedure. Both procedures accept a long integer range or long integer sequence as arguments. As with grid, ranges and sequences of other integer types are converted to have a base type of long):

```
V := 0.0 dim vector(1..3)
M := 0.0 dim matrix(1..4,1..6)    ! 4 rows, 6 columns
```

Vector and matrix domains have cyclic connectivity along all dimensions.

A vector domain conforms to a second vector domain if both contains the same number of elements. It will also conform to a one dimensional grid containing the same number of elements.

A matrix domain conforms to a second matrix domain if both domains have the same number of elements associated with each dimension. A matrix domain also conforms to a two dimensional grid domain with the same number of elements along each dimension.

iv. Domain of a sequence or tuple of sequences.

The domain of a sequence, block sequence or integer range *s* is given by **grid(0l..size(*s*)-1l)**

The domain of a cyclic sequence *c* is given by **grid(cycle(0l..size(*c*)-1l))**

The domain of a tuple of sequences **[s_1, cycle(s_2), …]** is given by **grid(0l..size(s_1)-1l,cycle(0l..size(s_2)-1l), …)** generalised to any combination of cyclic and non-cyclic sequences.

v. Domain shapes

All domains have a ***shape***, obtained using the procedure call **shape(*d*)**,

The shape of a grid is a tuple of long integer values. Each value is equal to the number of elements along the corresponding dimension.

```
shape(grid(1..3,2..6))==[3l,5l]
```

All valid domain shapes must themselves have a domain. The domain of a grid, vector or matrix shape (tuple of long integers) is a grid value with indices starting from zero:

```
dom( [ 3l , 5l ] )==grid( 0..2, 0..4 )
```

13. Arrays, Vectors and Matrices

i. Dimensions

A dimension value is created using the **dim** operator. This value contains the information needed to create and initialise an array. The left operand of the operator specifies an initialising value while the right operand provides the required domain.

```
        expr dim expr
```

e.g.: `0.0 dim grid(0..3,0..3).`

A distributed dimension value conforms to the type **dim{***element*,*domain***}**.

For convenience, all of the procedures applicable to constant arrays are also applicable to dimension values.

ii. Arrays

An array value stores a set of values corresponding to each element of a given domain.

Array values may be created by creating an object (usually a variable) initialised by a dimension value.

```
        A := 0.0 dim grid(0..3,0..3).
```

This works even if the dimension values are embedded in a larger structure.

```
      R := rec {array=1 dim dom, mask=false dim dom} where dom=grid(1..3,1..3)
```

Array types are defined using:

```
        [ type ] '[' opttypelist ']'
opttypelist::=
        [ type ] { ','[ type ] }
```

An array value conforms to array type T[U] if its element type conform to T and its domain conforms to U. If more than one type is present within the square brackets then the domain constraint is equal to a grid type parameterised by the supplied type list. For example:

```
int[grid2d]      ! Integer valued array over integer domain
int[]            ! Integer valued array over any domain
[]               ! Any array over any domain
int[,,]          ! Integer array over any 3d gridded domain -
                 !   - equivalent to int[grid{,,}]
int[,cycle]      ! Integer array over gridded domain with a cyclic dimension
                 !   - equivalent to int[grid{,cycle}]
```

One and two dimensional array values, with domains **grid{}** and **grid{,}** respectively, may be defined using the following syntax:

```
        '{' list2d '}'
list2d::=
        exprlist {';' exprlist } [';']
```

For example:

```
array_1d := { 1, 2, 3 }                        ! domain is grid(0..2)
array_2d := { 1, 2, 3 ; 4, 5, 6; 7, 8, 9}     ! domain is grid(0..2,0..2)
```

This notation creates an unnamed constant array object.
Array assignment cannot change the type of the array, but can change the value of its domain. It is therefore possible to resize an array through assignment. Arrays with zero size domains are permitted and will have no elements.

iii. Subscripts and slices

Subscript expressions have the following syntax:

'**[**' [*expr*] { '**,**' [*expr*] } '**]**'

Subscript expressions index an element or slice with reference to the domain of the array value being subscripted.

For grid, vector and matrix domains, with *N* dimensions, subscript expressions may consist of *N* expressions yielding integer values, integer ranges and integer sequences, or a single *N*-dimensional tuple of such values.

The use of all integer subscripts (or a tuple of integers) yields a single element of the array:

```
a:= 0 dim grid(1..6,1..9)
…
b:=a[1,3]
c:=a[s] where s=[2,2]
```

Subscripting an array with integer ranges or sequences yields a slice (a reference to a subarray).

```
d:=a[1..3,2..6 by 2]
```

A slice subscript computes a sub-domain by combining the domain of the value being slices with the subscript values as follows:

1. A value with an *N*-dimensional grid domain must be sliced by *N* range or sequence values, an *N*-dimensional tuple of sequence values or a single N-dimensional grid value.
2. A value with a vector domain must be sliced using a single range or sequence value, a one-dimensional tuple containing a one dimensional range or sequence value, a vector domain value or a one dimensional grid value.
3. A value with a matrix domain must be sliced using two range or sequence values, a two-dimensional tuple containing range or sequence values, a matrix domain value or a two dimensional grid value.
4. A slice value that is a grid, vector or matrix domain creates a slice over that value, which should be a sub-domain of the domain of the value being sliced (or a subdomain of a vector, matrix or grid conforming to that domain). Thus for a slice *x*[*y*], **dom(***x***) includes** *y* must be true.
5. A slice value that is an integer range *low***..***high* restricts indices along the sliced dimension to those lying in *low≤x≤high*.
6. A slice value that is an integer sequence *low***..***high* **by** *step* restricts indices along the sliced dimension to those lying in *low≤x≤high* and multiplies the interval between indices by the factor *step.*
7. A slice value that is an open integer range **…***high* restricts indices along the sliced dimension to those lying in *x≤high*.
8. A slice value that is an open integer range *low***…** restricts indices along the sliced dimension to those lying in *low≤x*.

9. A slice value that is a stride definition **by** *step* multiplies the interval between indices by the factor *step.*
10. A **null** slice value does not restrict or modify indices along the given dimension.

In the above example, the domain of `d` would be `grid(1..3,2..6 by 4)`.

If a subscript expression contains a mixture of integer and range or sequence values, the resulting slice will have a reduced number of dimensions:

```
e:=a[1..3,5]
```

When open ranges are used in subscripting, they act as if they were a closed range with the upper or lower bound as appropriate taken from the domain of the array:

```
f:=a[...7,2...]  ! Yields a slice of dimension grid(1..6,2..9)
```

A missing or null subscript is replaced by sequence from the corresponding dimension in the array domain, effectively selecting the entire dimension:

```
g:=a[2,]         ! Yields a slice of dimension grid(1..9)
```

An element or slice of an array may appear in an object element reference (left hand side of an assignment or **&** argument). In an assignment, a slice on the left hand side may be set to a conforming array value or to a single value which is set to all elements of the slice.

```
h[,2]=0
m[2,2]=m[2,2]+1
n[0..2,0..2]=m[3..5,3..5]
f(&m[0..5,])
```

iv. Vectors and matrices

A vector is an array defined over a **vector** domain.

A matrix is an array defined over a **matrix** domain

Vector and matrix values may be created using a generating expression:

```
          '('list2d ')'
list2d::=
          exprlist {';' exprlist } [';']
```

For example:

```
 v :=  ( 1, 3, 1 )

 a : = ( 3, 1, 1
         0, 2, 1
         0, 0, 1 )
```

As with array generators, all elements must have the same type (numeric type balancing is not invoked.) Note the use of the optional semicolon rule in the above example.

Matrix and vector values follow the usual rules for matrix multiplication using the '*****' operator. Vectors act as row or column matrices as appropriate in the context of matrix multiplication.

14. Distributions

i. Overview

A ***distribution*** splits a domain shape into a set of tiles, each tile corresponding to one element of a topology.

ii. Tiles

A tile describes a subset of the domain of a shape. There are two types of tiles. One is a grid with range or sequence (but not cyclic sequence) elements. An *N*-dimensional tile conforms to the type **tile*N*d** and to the overall type **tile**.

The second type of tile is the ***nested grid***. A nested grid consists of a base grid and a grid of offsets. The tile consist of the set of points by applying all offsets to each point in the base grid. A nested grid is created using **nest(***offsets*, *base***)** and conforms to the type **nest{** *offsets, base* **}**. A nested grid value is sequentially iterable (in a **for each** statement), with the offsets varying more rapidly than the base values.

iii. Block distribution

A ***block distribution*** divides an *N*-dimensional shape [$D_1, \ldots D_N$] over a topology [$T_1, \ldots T_N$] by partitioning each dimension *i* into T_i tiles: floor($j*D_i/T_i$)..floor($(j+1) \cdot (D_i-1)/T_i$), $j \in 0..T_i-1$.

A block distribution value is defined using the **block(***shape*, *topology***)** procedure:

```
    distr:= block( [ 91, 91] , [ 31, 31] )
```

Block values conform to the type **block*N*** where *N* is the number of dimensions and to the generic type **block**.

iv. Fixed block distribution

A fixed block distribution divides an *N*-dimensional shape [$D_1, \ldots D_N$] into fixed-size blocks sized [$B_1, \ldots B_N$] over topology [$T_1, \ldots T_N$]. Along a given dimension *i*, a valid block size B_i must satisfy $B_i \cdot T_i \leq D_i < B_i \cdot (T_i+1)$. Along this dimension, tile *j*, $j \in 0..T_i-1$ is defined by $j \cdot B_i$ .. $(j+1) \cdot B_i$

A fixed block distribution value is created using the **fblock(***shape*, *blocksize*, *topology***)** procedure

Fixed block values conform to the type **fblock*N*** where *N* is the number of dimensions and to the generic type **fblock**.

v. Cyclic distribution

A cyclic distribution divides an *N*-dimensional shape [$D_1, \ldots, D_N$] into cyclically varying points over topology [$T_1,\ldots,T_N$]. The tile for a given element [$J_1 \ldots J_N$], $J_i \in 0..T_i-1$, is given by grid(J_1 .. D_1 by T_1, … J_N .. D_N by T_N).

A cyclic distribution value is created using the **cyclic(***shape*, *topology***)** procedure.

Cyclic values conform to the type **cyclic*N*** where *N* is the number of dimensions and to the generic type **cyclic**.

vi. Block cyclic distribution

A ***block cyclic distribution*** divides an *N*-dimensional shape [$D_1, \ldots D_N$] into cyclically varying fixed-size blocks of size [$B_1, \ldots B_N$] over topology [$T_1, \ldots T_N$]. The tile for a given element [$J_1 \ldots J_N$], $J_i \in 0..T_i-1$, is given by nested grid: nest(grid(0..B_1-1, .., 0.. B_N-1) , grid($J_1 \cdot B_1$ .. D_1 by B_1, … , $J_N \cdot B_N$ .. D_N by B_N))

A block cyclic distribution value is created using the **block_cyclic(***shape*, *topology, blocksize***)** procedure:

Block cyclic distributions conform to the type **block_cyclic*N*** where *N* is the number of dimensions and to **block_cyclic**.

vii. The distribute procedure.

The **distribute** procedure is an alternative means of creating distribution values that is used by parallel statements. A call to distribute has the form: *distribution*:=**distribute(***partition*,*domain*,*topology*,*blocksize***)**. Here partition is a value indicating the type of distribution to create. The following variants of **distribute** are defined:

```
proc distribute(part:$block,dom,topo,block:null)=block(shape(dom),topo)
proc distribute(part:$fblock,dom,topo,block)=fblock(shape(dom),topo,block)
proc distribute(part:$cyclic,dom,topo,block:null)=cyclic(shape(dom),topo)
proc distribute(part:$block_cyclic,dom,topo,block)=
                                      blockcyclic(shape(dom),topo,block)
```

This mechanism enables programmer-defined distributions to be used by PM parallel statements by defining new variants of the distribute procedure. Note that the partition value does not have to be a name – more complex values may be used, possibly encoding distribution parameters beyond a simple block size.

15. Parallel execution

i. The for statement

The **for** statement executes its body concurrently for every element of one or more domains or arrays:

> **for** *iter subexp* [**using** *keyargs*] [**conc**] *loopbody* **endfor**
>
> *iter*::=
>
> *name* **in** *exp* { **','** *name* **in** *exp* }
>
> *loopbody*::=
>
> [**do** *statements*][**return** *definitions*]

For example:

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

It is possible to iterate over more than one domain and/or array, providing they all share the same domain shape:

```
    for pixel,pixel2 in image1, image2 do
        if pixel+pixel2>threshold then
            pixel=pixel-threshold/2
            pixel2=pixel2-threshold/2
        endif
    endfor
```

Note that concurrent execution in this context does not necessarily imply the use of separate threads (of whatever kind) for each invocation of the loop body. A for loop may be transformed into a sequential loop through code transformation and does not necessarily imply overhead over and above maintaining concurrent copies of some variables local to the statement.

Statements in the body of a parallel loop are not normally allowed to modify variables defined outside of the scope of the **for** statement. There are two exceptions to this rule. Firstly, if an item being iterated over is an array, then an assignment to the iteration variable will modify the value in the array. The assignment to the outer array variable comes into effect at the end of the **for** statement.

```
    for i,j in array,dom(array) do
        i=i+1
        i=array[j]+i*2    ! array still has original value
    endfor                 ! values in array now updated
```

ii. Returning values from a for statement

A **for** statement may return one or more loop invariant expressions using the **return** clause:

```
for point in domain do
    value:= f(point)
return
    sum_f:=sum::value
endfor
```

Loop-invariant expressions are described in a later chapter. Declarations in the **return** clause create objects in the scope enclosing the **for** statement.

iii. Parallel Search

The **find** statement executes its body for an arbitrary number of elements of a given iterable expression (or expressions) until it completes at least one successful invocaton for which the expression in the **when** clause yields **true**. Values computed by an arbitrary successful invocation are used to generate one or more objects that are created by the **return** clause in the scope enclosing the **find** statement.

> **find** *iter* [*subexpr*] [**using** *keyargs*] [**conc**] [**do** *statements*] *found* **endfind**
> *found*::=
> **when** *xexpr* **return** *definition* [*subexpr*] *finddefault* { '**;**' *definition* [*subexpr*] *finddefault* }
> *finddefault*::=
> **default** ('**(**'*exprlist* '**)**' | *expr* | *call*) [*subexpr*]

For example, the following code returns an arbitrary pixel value and location for which the pixel value exceeds a given threshold.

```
find pixel_value, pixel_loc in image, dom(image)
when pixel>threshold
return
     loc_above_threshold := pixel_loc default [-1,-1]
     value_above_threshold := pixel_val default -1
endfind
```

The right-hand-side expressions in the **return** clause are computed for an arbitrary loop invocation for which the **when** expression evaluates to **true**. Unlike **return** expressions in a **for** statement they do not have to be loop-invariant. If no loop invocation yield a **true** value for the **when** expression then expressions following the **default** keyword are used to generate the returned objects.

If a **return** clause contains multiple left hand sides, then the default clause must consist of either a procedure call capable of returning the correct number of values or a comma separated list supplying the required number of values enclosed in brackets:

```
return x,y:=get_values(z)  default (0,0)
```

Values in the default clause may not refer to objects defined in the body of the **find** statement. The right hand side of the **return** clause and the default expression may both have subexpressions. These subexpressions are independent of each other:

```
x:=0
find i in … do … when …
return y := a+b where a=f(i),b=g(i) default c + d where c=h(x),d=u(x)
endfind
```

The following return clause would cause an error.

```
return q:=f(i)+r default r**2
                where r=v(x)     ! r is not available outside of
                                 ! the default clause
```

The types of the right-hand side and default expressions must be indentical.
The body of a **find** statement may not contain communicating operations that take channel arguments, except within nested parallel **for** or **do also** statements.

iv. Task parallelism

Task parallelism is supported by the **do** .. **enddo** construct.

```
        [ using keyargs ] [ with definitions ] do rstatements { also do rstatements } enddo
rstatements::=
        statements [ return definitions ]
```

The semantics of a **do** statement are defined according to an equivalent **for** statement.

using *opts* **with** *statements*$_0$ **do** *statements*$_1$ **also do** *statements*$_2$ … **also do** *statements*$_N$**enddo**

is semantically equivalent to:

```
for _index in grid(0..N-1)
using opts
do
        statements_0
        select _index
        when 0 do statements_1
        when 1 do statements_2
        …
        when N-1 do statements_N
        endselect
endfor
```

As indicated by this equivalence, a **do also** statement may contain communicating operations.

16. Processor allocation

i. Tasks, processors and processor ownership

PM programs are assumed to be executed on a set of ***processors***. Processors are capable of efficiently executing a set of ***tasks***. The PM specification does not define how to translate the abstract concept of a processor into hardware: a processor can relate to anything from a single core to a sub-cluster. The primary requirement is that a processor is capable of keeping its own hardware busy running available tasks which effectively requires efficient mechanisms for task migration within the processor. Task migration between processors, however, is assumed to either introduce a degree of overhead which the programmer has to take into account, or not to be possible at all in any acceptably efficient manner. Tasks are a much lighter weight concept than threads (or even light threads) – in many cases a task may correspond to no more than one iteration of a loop in the final implementation of the PM code.

Each PM task is said to ***own*** a set of processors. Depending on the implementation, this may be interpreted in one of two ways which are equivalent in terms of PM semantics (if not necessarily in terms of performance):

1. The task is running simultaneously on every processor in its owned set.
2. Every processor in a task's owned set is available to the task to create child tasks.

Ownership of a processor can be shared among tasks. In this case, the processor will be responsible for scheduling the tasks that own it.

PM assumes that the number of processors available to the program is fixed by the implementation. When a PM program proceeds, this owned processor set is divided up as follows:

i) On start-up, the PM program runs a single task. This task owns all available processors.

ii) When a parallel statement (**for**, **find** or **do**.. **also do**) is encountered, a ***processor topology*** is computed using the statement's domain and information in the **using** clause (if present). Elements of the domain are ***partitioned*** over this topology.

    (a) A topology is a set of processors with (optional) information concerned with their preferred connectivity.
    (b) A domain is partitioned by assigning sub-domains to each processor in the topology.

iii) A new task is created for each element in the domain and assigned (possibly shared) ownership of a processor in the topology according to its position in the partition.

iv) If the topology does not contain every processor owned by the parent task then ownership of the remaining processors is given to some or all of the newly created child tasks according to a ***work-sharing*** algorithm.

    (a) The work-sharing algorithm associates each processor *S* not in the topology with a processor *P* that is in the topology. Any task that owns *P* is then given (possibly shared) ownership of *S*.

v) The parent task is suspended and all newly created child tasks are executed. Once all of the child tasks have completed, ownership of the processors is returned to the parent task which is then resumed.

ii. Concurrent clause

It is also possible to specify that a **for** or **find** statement is concurrent only:

for … **conc do** …. **endfor**

This form of the **for** statement does not partition its tasks among available processors. Each processor owned by the parent task runs the child tasks for all domain elements with the distribution set to **null**. All nested statements within the dynamic scope of this statement (including those in called by procedures) are also concurrent, whether not they include a **conc** keyword (**using** clauses are ignored).

iii. The using clause

The partitioning and work-sharing processes may be modified by providing keyword arguments in the **using** clause of a **for**, **find** or **do** statement. The following options are defined.

topo=	Use the given topology in place of the system default.
part=	Set the value used to select the partition algorithm.
block=	This option sets the block size used by the partition procedure. Usually a tuple of long integer values.
wshare=	Set the value used to select the work sharing algorithm
work=	Specifies an amount of work associated with each domain element. Used by worksharing algorithms. Should be a long integer array over the domain of the parallel statement.

iv. Topology

If **topo=** is not present, then the default topology for an *N*-dimensional domain is determined by factoring the number of processors owned by the current task to yield a Cartesian topology (tuple of long integer values) with as close as possible to the same number of elements along each dimension.

If **topo=***topology* is present then the given value is used for the topology.

v. Partitioning

If the current task only has ownership of a single processor, then no partitioning is necessary or possible and the distribution is set to **null**.

If more than one processor is owned by the current task, then the partitioning algorithm calculates a distribution based on the domain and topology associated with the parallel statement. The default algorithm (**part=** not present) works as follows:

If **block=** is not present then a block distribution is used for grid domains and a block cyclic distribution with a default block size used for matrix and vector domains.

If **block=** is set equal to a tuple of integer values then a fixed block distribution is used for grid domains and a block cyclic distribution used for matric and vector domains, both with the given block size.

If **part=***partition* is present, the **distribute** procedure is used:

*distribution***:=distibute(***partition*,*domain***,***topology***,***block***)**.

If **block=** is not present then it is set to null in this procedure call.

vi. Work-sharing

If the **wshare=** option is absent then the ***default work-sharing algorithm*** is used:

The algorithm first assigns processors i∈[1..*N*] not in the topology to tasks j∈[1..*M*] associated with elements x_j of the domain associated with the parallel statement in sequential iteration order.

1. If the **work=***A* option is present in **using**, then for each processor *i*, a domain element number *j* is calculated by solving:
$$\sum_{k=1}^{j-1} A[x_k] < \frac{i}{N+1}\sum_{k=1}^{M} A[x_k] \leq \sum_{k=1}^{j} A[x_k]$$
2. If **work=** is not defined then processors are allocated as if **work=***B* was present, and *B***[***x***]=1** for every element *x* of domain *D*.

Once the task number *j* has been calculated, the processor *p* associated with task *j* is determined. Each task that owns *p* is then also given (possibly shared) ownership of *i*.

If the **wshare=***workshare* option is present, then processor number *p* in the topology is determined for processor number *i* not in the topology using:

*p***=workshare(***workshare***,** *domain***,** *distribution***, size(***topology***),** *i***,** *N***,** *work*)

Each task that owns *p* is then also given (possibly shared) ownership of *i*.

Programmer-defined variants of the **workshare** procedure may be used to create new work sharing schemes.

17. Communicating operations

i. Overview

The **for** statement enables each invocation of its enclosed body of statements to be executed concurrently and possibly in parallel. Numerical models will usually require interaction between tasks processing different elements of the model domain. In PM this communication is enabled by providing a range of operators that view a value local to each task as if it were an array defined across all sibling tasks, each element corresponding to the local value in a single task. This is primarily achieved using '**{}**' operators. *x*%**{***subs***}** enables access to and update of the variable value associated with any given task (or a slice of such values). The binary operator *x***@{***ndb_desc***}** returns values from the local neighbourhood of the calling task. Arrays (both standard and distributed) defined outside of the parallel statement may be accessed and modified using *x***{***subs***}.**

ii. Invariant values

Communicating operations take account of whether a given value is ***invariant***. An invariant value is the same in each invocation and communication operations involving such values are typically more efficient. An invariant value is:

1. A literal value
1. A variable or constant defined outside of the enclosing **for** statement
2. The result of a procedure call for which all arguments are invariant
3. A constant defined inside the **for** statement which is set to an invariant value
4. A subexpression in a **where** clause that has been set to an invariant value.
5. A parameter to a communicating procedure with the **invar** attribute.

iii. Channel variables

Channel-based communicating operations provide a view of a given value across all invocations of the enclosing **for** or **do also** (but not **find**) statement. These values must be stored in channel variable. A channel variable is any variable defined within the body of **for** statement or **with** clause of a **do also** statement, but not within any **if** or **select** statement nested within the **for** statement body or **with** clause (or within any statement nested within such a conditional statement). For example:

```
for … do
      x:=0
      if … then
            y:=0
            while … do
                  z:=0
                  ! Here x is a channel variable, y and z are not
            endwhile
      endif
      while … Do
            w:=0
            ! Here both w and x are channel variables
      endwhile
endfor
```

iv. Channel subscript: %{}

The channel subscript operator enables a channel variable to be subscripted as if it were an array defined over the domain associated with the enclosing parallel statement. For example, the following code should not yield an error:

```
for i in array do
      for each j in dom(array) do
            check array[j]==i%{j}
      endfor
endfor
```

Channel subscripts may also be used to update specified channel variables:

```
D:= 0 dim grid(1..N)
…
for i in 1..N, j in D do
      if i>1 then
        j%{i-1}=j
      endif
endfor
```

v. Neighbourhood subscript: @{}

The neighbourhood subscripting operator **x@{***ndb_desc***}** returns values a channel variable 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
```

The neighbourhood is defined with respect to the controlling domain of the enclosing for statement, shifted so that its zero point lies at the domain element associated with the current invocation.

As with conventional array subscripting, the result of an **@** operator may be a single value or a slice. In contrast to conventional subscripting, these are treated in different ways in order to cater for edge effects. Unless a domain exhibits cyclic connectivity in all dimensions, the local neighbourhood of some elements in the domain will contain elements outside the domain itself. Identification of these off-edge elements is achieved in one of two ways:

1. If the neighbourhood description yields a single element of the controlling domain, then the local value associated with that element is returned as an optional value which will contain a null value for points over the edge of the controlling domain
2. If the neighbourhood description yields a slice (including a slice containing one element) then the extent of that slice will be clipped by the edge of the controlling domain. If the neighbourhood requested for a given element lies completely outside of the controlling domain, then the returned slice will have zero elements. For example, within the one dimensional domain grid(1..5) the neighbourhood [-1..1] will yield a slice covering [0..1] at point 1, [-1..1] at points 2..4 and [-1..0] at point 5.

As an example of a neighbourhood that is a slice, the following code implements a mean filter:

```
for pixel in image do
      filtered_pixel=sum(n)/size(n)
            where n=pixel@{-1..1,-1..1}
return
      filtered_image=gather::filtered_pixel
endfor
```

vi. Communicating array subscript {}

A communicating array slice allows access to an element or slice of an array defined outside of the parallel statement (but within any outer parallel statement within which the current parallel statement is nested). The array may be a conventional array or a distributed array. Note that for a conventional array, the semantics are essentially the same as using a regular array subscript using **[]** - although using the communicating version allows a distributed array to be substituted at a future point in program development.

```
A := 0 dim grid(1..N)
sum:=0
for i in 1..N do
      A{i}=i**2
      sum=sum+a{i}
endfor
```

18. Distributed Arrays

i. Overview

Distributed arrays explicitly store different groups of elements on different processors. The use of channel variables covers many common cases where distributed arrays would be needed in other parallel languages. However, there will remain occasions where it will be necessary to maintain state between parallel statements or to simultaneously operate on values with non-conforming domains.

ii. Distributed dimensions

Distributed dimension values are similar to dimension values – they provide a template proving the information to create a distributed array. The array itself is created when an object (or sub-object) is initialised to a distributed dimension value. The generating expression for a distributed value has the following syntax:

distr '**{**' *expr* [*usingclause*] '**}**'

If the expression is a dimension value then the resulting distributed dimension value contains the complete information necessary to create a distributed array.

If the expression is a domain, then the resulting incomplete distributed dimension value may be converted into a complete distributed dimension value by using it as the right operand of a **dim** operator.

If a distributed dimension value (complete or incomplete) is used as the right operand of a dim operator, the result is a complete distributed dimension value with the given initial element value.

The **using** clause in a distributed dimension value generating expression may contain the following:

align=	Use a given domain to compute the partition rather the array domain. The array domain must be a subdomain of the given domain.
topo=	Use the given topology in place of the system default.
part=	Set the value used to select the partition algorithm.
block=	This option sets the block size used by the partition procedure. Usually a tuple of long integer values.

The distribution of array tiles is obtained using the same partitioning algorithm as for parallel statements applied to the processors owned by the current task. However, no work-sharing procedure is employed – processors not in the topology are assigned a tile of zero size, as are any processors covering parts on an alignment domain that does not intersect with the domain of the distributed array.

iii. Parallel scope

A parallel scope is defined as including all statements within a given parallel statement but not within any parallel statement nested within that statement. All statements not within any parallel statement form the outermost parallel scope.

iv. Distributed array creation and access

A distributed array is created by initialising an object (or sub-object) to a distributed dimension value. A distributed array may also be created by initialising an object to a distributed array created within the scope of the same parallel statement (alternatively arrays may be defined outside of the scope of any parallel statement).

A distributed array may not:

1. form an element value in an array or distributed array
2. be returned from a parallel statement to an outer parallel scope
3. be referenced from an inner parallel scope, except in limited circumstances

Within a parallel scope immediately nested within the one in which a distributed array is defined, it is possible to:

1. Use the distributed array as a regular argument corresponding to an **invar** parameter of a loop procedure.
2. Use the distributed array a an operand to a communicating array slice *x*{*subs*}.
3. Obtain the elements of the distributed array held on the current processor using *x*.**local**

At any point within the scope of a distributed array it is possible to obtain array information using *x*.**distr**. This returns the following record: **rec ddom{dom=***domain*,**dist=***distribution*,**align=***alignment_array*,**tile=***local_tile***}**.

v. Distributed arrays in for and find statements

Distributed arrays may be used in parallel **for** and **find** statements.

A distributed array conforms to another distributed array if:

1. Both have null alignments and they have conforming domains
2. They have the same alignment and domain

A domain or array conforms to a distributed array with a null alignment if it conforms to the domain of that distributed array.

A parallel **for** or **find** statement operating over distributed arrays must:

1. Not have a **conc** keyword
2. Not have a **using** clause.

Note that a distributed array defines a tile for each processor (which may have zero size in some cases). Therefore each child task of a parallel statement operating over a distributed array will own only a single processor – all dynamically nested parallel statements will therefore effectively be concurrent.

19. Communicating procedures

i. Overview

Communicating procedures are user defined communicating operations that, unlike regular procedures, may incorporate other communicating operations.

proc *name* ('**%**' |'**::**') *params* [*special*] '**=**' *xexprlist* [**do** *statements* **endproc**]
proc *name* ('**%**' |'**::**') *params* [*special*] [*proccheck*] **do** *statements* [**result** '**=**'*xexprlist*] **endproc**
special::=
  reduce '**(**' *namelist* '**)**' | **local** | **global** | **each** '**(**' *namelist* '**)**'
params::=
  [*largs*] [*cpars*] ['**(**' [*pars* ['**,**' *keyparams*] | *keyparams*]] '**)**']
cpars::=
  '**[**' *name* ['**:**' *type*] { '**,**' *name* ['**:**' *type*] }'**]**'
pars::=
  { *param* '**,**' } (*param* | **arg** '**...**')
param::=
  ['**&**'] *name* ['**:**' [**invar**] *type*]

Communicating procedures can only be called inside **for**, **find** or **do also** statements. A call has the following syntax:

  name '::' [*cargs*] ['**(**' [*arglist*] '**)**'] ['**@**' *slice*]
  name '%' [*cargs*] ['**(**' [*arglist*] '**)**']
cargs::=
  ['**&**'] *name* | '**[**' ['**&**'] *name* { '**;**' ['**&**'] *name* } '**]**'

For example:

```
proc mean_filter%[x](n)=sum(x)/size(x)
      where nbd=x@{-n..n,-n..n}
proc image_mean::(x)=sum::x/count::x

for pixel in image do
      pixel = mean_filter%pixel(1)
      norm_pixel = pixel / image_mean::pixel
endfor
```

Communicating procedures that return a different value to each task use the '**%**' notation. Procedures that return the same value to every task use the '**::**' notation.

Parameters of communicating procedures fall into two categories *channel* parameters are places between square brackets while regular parameters are placed between parentheses. When the procedure is called, channel parameters must be provided with the name of a channel variable. Channel parameters may be passed as channel arguments to other communicating operations. Regular arguments to a communicating procedure and local variables defined within the communicating procedure may not be used as channel arguments to communicating operations.

It is possible to require a regular parameter to be loop-invariant using the **invar** keyword. A definition with **invar** parameters is strictly more specific than a definition where no parameter has this constraint. This facility enables special cases where one or more parameters are loop-invariant to employ specific algorithms. Keyword arguments in a communicating procedure call must always be loop invariant.

ii. Implicit arguments to communicating procedures

largs::=
    '**<**' *larg* { '**,**' *larg* } '**>**'
larg::=
    '**this_dom**' '**:**' *type* | '**this_distr**' '**:**' *type* | '**this_tile**' '**:**' *type* | '**this_index**' '**:**' *type*

Communicating procedures are automatically passed three additional arguments: **this_dom**, **this_distr** and **this_tile**, indicating the domain, distribution and tile of the enclosing parallel statement. These three parameters are always accessible within a communicating procedure. However, it is possible to specify them in the parameter list of a procedure definition in order to constrain their types. If specified, the parameters must be in the stated order and precede channel parameters. Type constraints on these parameters take a full part in procedure selection (by default they are unconstrained).

```
proc block_comm%<this_distr:block>[x]= …
y:=block_comm%(x)
```

iii. Reduction procedures

proc *name* '**::**' *params* **reduce** '**(**' *namelist* '**)**' '**=**' *xexprlist* [**do** *statements* **endproc**]
proc *name* '**::**' *params* **reduce** '**(**' *namelist* '**)**' [*proccheck*] **do** *statements* [**result** '**=**'*xexprlist*] **endproc**

There are a number of specialised forms of communicating procedure definition. Reduction procedures combine values though the application of a binary operator.

```
proc sum:: (x) reduce(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. A reduction procedure definition must specify the same number of channel parameters as the number of values it returns. Moreover, it must specify the same number of reduction variables (in the **reduce** clause) and the number of values it returns.

iv. Local procedures

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

A second special form of communicating procedure is the processor-local procedure, defined using the **local** form of the procedure body. Each argument to a **local** procedure is passed as an array over the domain of the tile associated with the current processor node (**invar** parameters are passed unmodified.) Regular arguments to the local procedure have optional-values elements which indicate if the call to the communicating procedure was made by the task associated with a given element (value defined), or if the task called a **sync** statement (value undefined). A local procedure must have a **%** form – it will return the same value for points generated on the same processor, but over the whole controlling domain, the return value will vary from processor to processor.

v. Global procedures

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

Global procedures are similar to local procures, except that the procedure body is effectively executed as if it were not nested in the enclosing parallel statement. Each argument to a **global** procedure is passed as a distributed array over the domain enclosing parallel statement (**invar** parameters are passed unmodified.) Regular arguments to the global procedure have optional-values elements which indicate if the call to the communicating procedure was made by the task associated with a given element (value defined), or if the task called a **sync** statement (value undefined). A global procedure may have either a **%** or a **::** form – it will return the same value for every point in the domain.

20. Structured parallel communication

vi. Overview

A complete sequence of communicating operations must be specified along each possible path through the body of a parallel statement, even if a given path does not execute all of these operations. If the body contains a conditional statements, one branch of which contains a communicating operation, then it is necessary to indicate which point in the other (non-empty) branches correspond, in terms or synchronisation of values, to the communicating operation. This is achieved by providing a sync statement at those corresponding points.

vii. The sync statement.

sync *syncvars* { '**,**' *syncvars* }
syncvars::=
name | '**[**' *namelist* '**]**'

The sync statement indicates the synchronisation point associated with one or more communicating operations. For each operation being matched, a (possibly empty) set of channel variables must be provided.

```
 for in in 1..N do
     j:=f(i)
     if i<N/2 then
          j=j%{i*2}
          sync j
     else
          sync j
          j=j%{i/2}
     endif
endfor
```

viii. Synchronisation matching rules

Two basic blocks of non-control statements are ***sync-compatible*** if they contain matching sequences of communication operations and **sync** statements. When these operations and statements are considered in order for the two blocks, then a communicating operation must always be matched with a **sync** statement listing the same channel variables.

If a conditional expression (**if**, **select**) contains communicating operations then each non-empty conditional branch of the statement must be sync-compatible.

If communicating operations appear within non-parallel loops (**while**, **repeat**, **for each**) then these loops are not forced to repeat the same number of times for each task. If a communicating operation executed in a longer-running loop tries to access a channel value from a task where the loop has already terminated, the operation will be supplied with the value that the channel variable possessed immediately prior to the termination of the shorter-running loop. There is an implicit synchronisation at the end of any sequential loop containing communicating operations.

If a sequential loop containing communicating operations occurs in a conditional statement, then a loop containing a sync-compatible body of statements must occur at the same position (in terms of the sequence of communicating operations/**sync** statements) on each non-empty branch of the conditional statement. These matching loops do not have to employ the same statement forms – a **while** loop can match a **repeat** loop and both can match a **for each** loop. It is also permissible for the body of one loop to be sync-compatible with a fixed repetition of the body of the other loop.

ix. The if invar statement

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

On occasion it is necessary to select a different sequence of communicating operations based on a loop invariant expression. This cannot be achieved using a conventional if statement as this would break sync matching. The **if invar** statement is provided for these occasions.

21. Syntax

```
module::=
          program_module | library_module
program_module::=
          [ decls ';'  ] statements
library_module::=
          decls [';' debug statements enddebug  ]
decls::=
          { import ';' } ( import| decl )  { ';' decl }
import::=
          include modname  [ modifiers ]  { ',' modname  [ modifiers ]  }
modname::=
          name { '.' name }
modifiers::=
          '{' modifier { ',' modifier } '}'
modifier::=
          [ type  | param  | proc ] name '=>' name
          struct name
decl::=
          procdecl | typedecl | paramdecl
procdecl::=
          proc procname params [ each ]  '=' xexprlist [ do statements endproc ]
          proc procname params [ each ] [ proccheck ] do statements [ result '='xexprlist ] endproc
          proc procname ( '%' | '::' )  commpars [ special ]  '=' xexprlist [ do statements endproc ]
          proc procname ( '%' | '::' )  commpars  [ special ] [ proccheck ] do statements [ result '='xexprlist ] endproc
procname::=
          name | opname  | null
opname::=
          assnop  | not     | '[]'     | '{}'     | '@{}'  | '@{}|'
           '=='     | '/='     | '>'      | '>='     | in       | includes
           '#'       | '=>'     | dim   | '..'       | by
assnop::=
          '+'        | '-'        |'*'       | '/'       | '**'     | mod   | and     | or       | '//'
proccheck::=
          check exprlist [ whereclause ]
special::=
          global   | local    | each  | reduce '('  namelist ')'  |
each::=
          each '('  namelist ')'
params::=
           '('  [ pars [ ',' keypars ] | keypars ] ')'
commpars::=
           [ lpars ] (  cpars   | [ cpars  ] params )
pars::=
          { par ',' } ( par | arg '...' )
par::=
          [ '&' ] name[ ':' [ invar ] type ]
keypars::=
          { name '=' expr ',' } ( name '=' expr | key '...'  )
lpars::=
          '<'  lpar  ':' type  { ','  lpar  ':' type  } '>'
lpar::=
          'this_dom' | 'this_distr' | 'this_tile'
```

cpars::=
'**[**' ['**&**'] *name*['**:**' *type*] { '**,**' ['**&**'] *name*['**:**' *type*] } '**]**'
namelist::=
name { '**,**' *name* }
typedecl::=
type *name* ['**{**' *typeparams* '**}**'][**in** *namelist*] [**is** *typelist* | **includes** *typelist* | **also includes** *typelist*]
typeparams::=
name '**:**' *type* { '**,**' *name* '**:**' *type* }
typelist::=
type { '**,**' *type* }
type::=
name ['**{**' *opttypelist* '**}**']
any
null
'**$**' *name*
proc ['**{**' *procname* '**}**']
[*type*] '**[**' *opttypelist* '**]**'
'**<**' *type* '**>**'
[**struct** | **rec**] [*name*] '**{**' [**include**] *name* ['**:**' *type*] { '**,**' [**include**] *name* ['**:**' *type*] } '**}**'
opttypelist::=
[*type*] { '**,**' [*type*] }
paramdecl::=
param *name* '**=**' *xexpr*
statements::=
statement { '**;**' *statement* } ['**;**']
statement::=
if *xexpr* **then** *statements* { **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**
if invar *xexpr* **then** *statements* { **elseif invar** *xexpr* **then** *statements* } [**else** *statements*] **endif**
select [*xexpr*] { **when** *xexprlist* **do** *statements* } [**otherwise** *statements*] **endselect**
while *xexpr* **do** *statements* **endwhile**
repeat *statements* **until** *xexpr*
for each *iter subexpr* [(**while** | **until**) *xexpr*] **do** *statements* **endfor**
any *name* ['**=**' *xexpr*] *rstatements* **endany**
check [*string* '**:**' *expr* [*whereclause*]
debug *statements* **enddebug**
for *iter* [*subexpr*] [**using** *keyargs*] [**conc**] *rstatements* **endfor**
find *iter* [*subexpr*] [**using** *keyargs*] [**conc**] [**do** *statements*] *found* **endfind**
[**using** *keyargs*] [**with** *definitions*] [**conc**] **do** *statements* [*rtn*] { **also do** *statements* [*rtn*] } **enddo**
sync *syncvars* { '**,** ' *syncvars* }
assignment [*subexpr*]
definition [*subexpr*]
call [*subexpr*]
iter::=
name **in** *expr* { '**,**' *name* **in** *expr* }
rstatements::=
do *statements* [*rtn*]
rtn
rtn::=
return *defintion* { '**;**' *definition* }
found::=
when *xexpr* **return** *definition rtndefault* { '**;**' *definition rtndefault* }
rtndefault::=
default ('**(**' *exprlist* '**)**' | *expr* | *call*) [*subexpr*]
syncvars::=
name | '**[**' *namelist* '**]**'

```
definitions::=
        definition { ‘;’ definition }
definition::=
        name ( ':=' | '::=' ) xexpr
        deflhs ',' deflhs { ',' deflhs } ( ':=' | '::=' ) call
        vlhs '=' expr
        vlhs',' vlhs { ',' vlhs } '=' call
deflhs::=
        name | '_'
vlhs::=
         '_'     | var name [ ':' type ]   | var '{' namelist '}' [ ':' type ]   | const name [ ':' type ]
assignment ::=
        lhs '=' expr
        lhs',' lhs { ',' lhs } '=' call
lhs::=
        ref      | vlhs   | name '%' cslice
ref::=
        name [ qualifier ]
xexpr::=
        expr subexp
subexpr::=
        [ check exprlist ] whereclause
whereclause::=
        { where constdef { ','  constdef }  }
constdef::=
        name  '=' expr
        deflhs  ',' deflhs  { ',' deflhs  }  '=' call
xexprlist::=
        exprlist subexp
exprlist ::=
        expr { ',' expr }
expr::=
        lowest to highest precedence
        expr  '=>' expr '||' expr
        expr '//' expr
        expr '#' expr
        expr or expr
        expr and expr
        not expr
        expr [ '==' | '/=' | '>'| '<' | '>=' | '<=' | in | includes ] expr
        expr dim expr
        expr by  expr
        expr '..' expr  | '...' expr  | expr '...' | by expr
        expr [ '+' | '-' ] expr
        expr mod expr
        '-' expr
        expr [ '*' | '/' ] expr
        expr '**' expr
        expr '|' expr
        term  [ qualifier ]
```

term::=

name
literal
'**(**'*expr* '**)**'
'**[**' *exprlist* '**]**'
call
array
'**<**' [*type*] '**>**' *term*
[**struct** | **rec**] [*name*] '**{**' [**include**] *name* '**=**'*expr* { '**,**' [**include**] *name* '**=**'*expr* } '**}**'
distr '**{**' *expr* [*usingclause*] '**}**'
name cslice
name '*%*' *cslice*
name '**@**' *cslice*

literal::=

'**$**' *name*
'**$**' **true**
'**$**' **false**
proc '**{**' *procname* '**}**'
true
false
null
number
string

call::=

(*name* | **null** | **proc** '**{**' *procname* '**}**') '**(**' [*arglist*] '**)**'
(*name* | **proc** '**{**' *procname* '**}**') '**%**' [*carglist*] ['**(**' [*arglist*] '**)**']
(*name* | **proc** '**{**' *procname* '**}**' '**::**' [*carglist*] ['**(**' [*arglist*] '**)**'] ['**@**' *cslice*]

carglist ::=

['**&**'] *name*
'**[**' [['**&**'] *name* { '**,**' ['**&**'] *name* }] '**]**'

arglist::=

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

arg::=

'**&**' *ref* | *expr*

keyargs::=

name '**=**' *expr* { '**,**' *name* '**=**' *expr* }

qualifier::=

{ '**.**' *name* | *slice* | *cslice* | '**"**' *term* | **over** *term* }

slice ::=

'**[**' [*expr*] { '**,**' [*expr*] } '**]**'

cslice ::=

'**{**' [*expr*] { '**,**' [*expr*] } '**}**'

array::=

'**(**' *list2d* '**)**'
'**{**' *list2d* '**}**'

list2d::=

exprlist {'**;**' *exprlist*} [';']

22. Intrinsic procedures

i. Arithmeric operations

`-(x:num)`	Negate
`+(x:num,y:num)`	Add
`-(x:num,y:num)`	Subtract
`*(x:num,y:num)`	Multiply
`/(x:num,y:num)`	Divide
`**(x:num,y:num)`	Power
`mod(x:num,y:num)`	Modulo
`max(x:any_real,y:any_real)`	Maximum
`min(x:any_real,y:any_real)`	Minimum

1. The result type of these procedures is determined using ***numerical type balancing rules***.
2. The result of arithmetic overflow or underflow, division by zero or modulo zero is not defined by the language standard.
3. The modulo operation gives **mod(***a*,*p***)=***a-floor(a/p)*p*

ii. Numerical comparisons

`>(x:any_real,y:any_real)`	Greater than
`>=(x:any_real,y:any_real)`	Greater than or equal to
`==(x:any_real,y:any_real)`	Equal to
`/=(x:any_real,y:any_real)`	Not equal to

The result type of these procedures is **bool**.

iii. Numerical conversions

```
int(x:num)                     long(x:num)
int8(x:num)                    int16(x:num)
int32(x:num)                   int64(x:num)
int128(x:num)                  real(x:num)
double(x:num)                  real32(x:num)
real64(x:num)                  real128(x:num)
cpx(x:num)                     double_cpx(x:num)
cpx64(x:num)                   cpx128(x:num)
cpx256(x:num)
```

1. These procedures convert a numerical value to the type indicated by the name.
2. If a complex value is converted to a non-complex value then the real part is taken.

`xx,yy=balance(x:num,y:num)` Numerical balancing

The balance procedure returns *x* and *y* converted to the type obtained by applying numeric balancing to the types of *x* and *y*

iv. General comparisons

`==(x,y) check x?=y`	Equal to
`/=(x,y) check x?=y`	Not equal to

The result type of these procedures is **bool**.

v. Logical operations

`and(x:bool,y:bool)`	Logical and
`or(x:bool,y:bool)`	Logical or
`not(x:bool)`	Logical not

The result type of these procedures is **bool**.

vi. String operations

`//(x,y)`	Concatenate string
`#(x,n:any_int)`	Format value as string with of width n
`#(x,n:tuple{any_int,any_int})`	Format value as string width n.d1 using n.d2 decimal places
`string(x)`	Convert value to a string

1. The result type of these procedures is **string**.
2. Arguments to **//** are converted to **string** using the **string()** procedure.

vii. Array operations

`dim(x,y:dom)`	Spread value x over domain y to create an array value
`over(x[],y:dom)`	Values of array x over conforming domain y
`redim(x[],y:dom)`	Value of array x over domain y with same number of elements (paired off using respective iteration sequences of dom(x) and y.
`dom(x:[])`	Domain of array x
`sum(x:num)`	Sum
`prod(x:num)`	Product
`maxval(x:num)`	Maximum value
`minval(x:num)`	Minimum value
`allof(x:bool)`	All values true
`anyof(x:bool)`	At least one value true
`count(x:bool)`	Number of values true

viii. Type comparison

`same_type(x,y)`	Type comparison

1. The result type of this procedure is either **$true** or **$false**.
2. The result is **$true** if both arguments have the same concrete type (alternatively if both `z:=x;z=y` and `z:=y;z=x` could execute successfully). Otherwise the result is **$false**.

ix. Ranges and sequences

Procedure	**Action**	**Result conforms to**
`..(x:range_base,y:range_base)`	Create range	`range{t}`
`..._(x:range_base)`	Create infinite range below *x*	`range_below{t}`
`_...(x:range_base)`	Create infinite range up to *x*	`range_above{t}`
`by(x:range)`	Create a sequence	`seq{t}`
`by(x:seq)`	Multiply stride of sequence	`seq{t}`
`by(x:range_below)`	Create infinite sequence below *x*	`strided_range_below{t}`
`by(x:range_above)`	Create infinite sequence up to *x*	`strided_range_above{t}`
`cycle(x:bd_seq)`	Create cyclic range or sequence	`cycle{t}`

The base type of the result, *t*, is the determined by applying numeric type balancing to the types (or for ranges and sequences to the base types) of the arguments.

Procedure	**Action**	**Result conforms to**
`low(x:range_base,y:range_base)`	Lowest point in range	range_base
`high(x:range_base)`	Highest point in range	range_base
`in(x:range_base,y:seq)`	Is point in the range?	bool
`includes(x:range,y:range)`	Does one range include another?	bool
`low(x:range_base,y:range_base)`	Lowest point in sequence (independent of direction)	range_base
`high(x:range_base)`	Highest point in sequence (independent of direction)	range_base
`step(x:range_base)`	Step of sequence	range_base
`start(x:range)`	First point in sequence	range_base
`finish(x:seq)`	Last point in sequence	range_base
`in(x:range_base,y:seq)`	Is point in sequence?	bool
`includes(x:seq,y:seq)`	Does one sequence include another?	bool
`dom(x:any_seq)`	Domain of range or sequence	dom
`is_cyclic(x:any_seq)`	Is this a cyclic range or sequence?	bool
`size(x:any_seq)`	Number of elements in sequence	long

`expand(x:any_seq,y:any_seq)`	Expand sequence *x* by extent of *y*
`contract(x:any_seq,y:any_seq)`	Contract sequence *x* by extent of *y*
`convert(x:any_seq,y:range_base)`	Convert sequence to have same base type as *y*

x. Domains

`grid(arg...:grid_base)`	Create grid domain
`vector(rows:grid_base)`	Create vector domain
`matrix(rows:seq,cols:seq)`	Create matrix domain

`expand(x:dom,y:dom)`	Expand domain *x* by extent of *y*	dom
`contract(x:dom,y:dom)`	Contract domain *x* by extent of *y*	dom
`loc_in_shape(x:dom,y)`	Return point y in *x* as equivalent point in **dom(shape(*x*))**	index
`conform(x:dom,y:dom)`	Domain y conforms to domain x	bool
`size(x:dom)`	Number of elements in domain	long

xi. Distributions

`block(shape:tuple,topol:tuple)`	Create block distribution
`fblock(shape:tuple,topol:tuple,block:tuple)`	Create a fixed block distribution
`cyclic(shape:tuple,topol:tuple)`	Create a cyclic distribution
`block_cyclic(shape:tuple,topol:tuple, block:tuple)`	Create a block cyclic distibution

All arguments must be values with the same number of dimensions

xii. Processor grouping

`sys_nprc()`	Number of processors available to the PM program
`this_prc()`	Processor rank within set of processors used by current parallel statement
`this_nprc()`	Number of processors used by current parallel statement
`shared_nprc()`	Number of processors owned by current task
`is_par()`	Does current parallel statement use more than one processor? (**bool** result)
`is_shared()`	Does current task own more than one processor? (**bool** result)

1. Processor ranks are numbered from zero
2. All result values are long integer unless otherwise stated.

xiii. Communicating procedures

`sum::[x:num]`	Sum
`prod::[x:num]`	Product
`maxval::[x:num]`	Maximum value
`minval::[x:num]`	Minimum value
`allof::[x:bool]`	All values true
`anyof::[x:bool]`	At least one value true
`count:: [x:bool]`	Number of values true