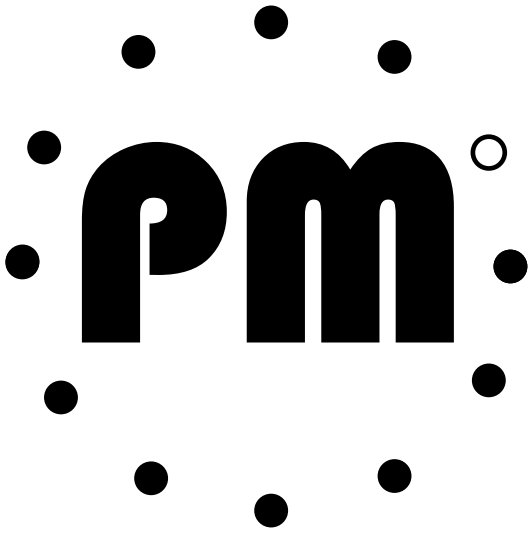

PM –Parallel Models Programming Language

Version 0.2

Revision 00

Language Reference (incomplete)

Contents

1. Introduction

i. Background

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

- Language structures facilitate the generation of vectorised and parallelised code
- Vectorisation and parallelisation should be explicit and configurable
- Programmers should be able to concentrate on coding the 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.

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).

ii. 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 '**||**' '**&&**' 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 most 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. The exception is the use of a semicolon to separate channel and non-channel arguments in a loop procedure call.

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:

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

Numeric constants take the following forms*:*

number	::=	[*integer_constant* | *real_constant* | *imaginary_constant*] [*bits*]
integer_constant	::=	[{ *digit* } '**r**'] { [*digit* | *letter*] } ['**l**']
real_constant	::=	{ *digit* } ['**.**'*digit* { *digit* }] ['**e**'['**+**' | '**-**'] digit { *digit* }]['**d**']
imaginary_constant	::=	[*integer_constant* | *real_constant*] '**i**'
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 '"'. They 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). 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* }

Names 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 with use

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

Modules may import declarations from other modules using a **use** statement which imports all parameters, types and procedures defined in the named module into the current module. However, the importing process does not import definitions indirectly accessed through **use** statements in the imported module.

A **use** statement must not result in two type or parameter definitions being accessible using the same name in the same module. Similarly, it must not result in procedure definitions accessed through the same procedure name in the same module having conflicting signatures. Name clashes may be avoided by using a qualifying clause in the **use** statement. This lists specific elements to import and optionally renames them:

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

```
use 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** or **proc** 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 and procedure selection are only be necessary in situations where specifically polymorphic values are generated (using ***polymorphic types*** or recursive procedure invocation). Implementing the PM type system within program analysis requires type analysis between and across modules.

PM programs operate on values.

- 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 $V \in T$ or for some type U, $V \in U$ and U conforms to T
- Type T conforms to type U if $T \in U$ or for some type V, $T \in V$ and V conforms to U

Both of these definitions may be applied recursively.

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. Affirm type

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

vi. 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*	
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**' and '**A**'..'**Z**' representing 10 to 36 respectfully (case insensitive).

```
123        2r101011101   16rdeadbeef
```

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

```
8898743248972847461                      ! long
255_8                                    ! int8
2r10101010101010101010101010101010_32    ! int32
```

vii. 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
```

viii. 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
```

ix. 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**	

x. 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.)

xi. Boolean type

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

xii. 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"
```

xiii. Proc type

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

```
proc cell_model
proc +
```

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

xiv. 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*] '**{**'*name* ['**:**' *type*] { '**,**' *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*] '**{**' *name* '**=**'*exp* { '**;**' *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. The structure of record value may optionally tagged with a further name. Component names are local to the structure or record. The name tagging the whole structure or record resides in a name space that is global to the program (unless preceded by an '_' in which case it is local to the given module)

A structure or record 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 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.

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 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 ('**&**')

xv. 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 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*.

xvi. 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. Unlike an optional polymorphic value, defined below, 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 '**^**' or '**|**' operators. Both return the value contained within the optional value if it is not null. The former returns an arbitrary value 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:=^y      ! Sets w to 2.0
x:=^z      ! Returns an arbitrary real value (may not be 2.0!)
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
```

It is possible to assign directly to the contents of an optional value:

```
^z=3.0      !  z now has the value opt(3.0)
```

An optional value **x** 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*.

xvii. 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*

The value contained within a polymorphic value may be obtained or changed using the unary '**^**' operator.
For example:

```
type int_or_string is int,string
a:= <int_or_string> 1
a= <int_or_string>"Hello"
print(^a)             ! Outputs "Hello"
^a=3
print(^a)             ! Outputs "3"
```

A polymorphic type constraint expression has the following form:

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

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

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

Type conformance for ^x where x has a polymorphic value may not be determinable at compile time. The value of ^x is said to have an ***ambiguous type***. Ambiguous types may require run-time analysis for type checking and procedure selection.

xviii. Optional polymorphic types

An optional polymorphic type is an extended form of polymorphic type which can hold any value conforming to a given type constraint and can additionally hold a **null** value. Applying the '**^**' operator to an optional polymorphic value containing a **null** value results in an error. Unlike an optional type, an optional polymorphic type will use a small fixed amount to storage to represent a null value. This enables the creation of indefinitely recursive data structures, such as linked lists and trees.

<opt *type>*

A value of an optional type may be created in a similar manner to that of a polymorphic type:

```
a:= <opt int> 1
print(^a) ! Will print the number one
a= <opt int> null
print(^a) ! Will generate an error – null values cannot be accessed
```

It is also possible to use the same operators as apply to optional types:

```
y:=<opt real> 2.0
z:=<opt real> null
a:=y|1.0         ! Sets a to 2.0
b:=z|1.0         ! Sets b to 1.0 (value to right of | is default)
print(isnull(y)) ! prints false
print(isnull(z)) ! prints true
^z=2.1           ! Changes value contained within the value of z
```

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. A scalable generic linked list would be declared as:

```
type linked_list is struct{head,tail:<opt linked_list>}
```

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. 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*)** return a value of one and **size(*r*)** returns **max(high(*r*)-low(*r*),0)** if *r* is a range of integer type.

ii. Open ranges

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 }
```

Open ranges are most commonly used in subscript expressions.

iii. 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 }

iv. 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**.

v. Generic sequence types

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

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

7. Domains, Arrays, Vectors and Matrices

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 (both sequentially and 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 themselves associated with a domain. The domain of a domain is known as its ***shape***. The domain of a shape *D* is always equal to itself: **dom(dom(*D*))==dom(*D*)** for any domain value *D*. Any domain *S* for which **dom(*S*)==*S*** is known as a ***shape domain***.

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 iterated over 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. 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.

iv. Arrays

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

Array values may be created using the **dim** operator, which creates an array with the same initialising value in each location:

expr **dim** *expr*

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

Array assignment requires that the two participating arrays have conforming domains. Arrays cannot change size – flexible size arrays must be implemented using polymorphic types. An array of type **int[]** cannot change size. A value of type **<int[]>** can refer to arrays of different sizes.

v. Vectors and matrices

A vector is an array defined over a **vector** domain – a specialised one dimensional domain.

A matrix is an array defined over a **matrix** domain – a specialised two dimensional 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.

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
```

Both vector and matrix types 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.

8. 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 '**&**'. A procedure call can only use & arguments if it does not itself form an argument for another procedure call or an argument for an operator and is not located in a **where** clause.

```
x,y=process_next_pair(&list)  ! Permitted
x=process_next(&list)*2       ! Not permitted – call is argument to *
                              ! operator
x=f(process_next(&list))      ! Not permitted – call is argument to
                              ! another call
if not get_next(&q,&v) then   ! Permitted
    print("At end")
endif
y=f(x)                        ! Not permitted (where clause)
  where x=process_next(&list)
```

iii. Variable-length argument lists

A procedure may accept a variable number of arguments:

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

From the body of the procedure, the excess arguments are accessed using a ***variable argument access expression***: **arg[]** and **argc**. **argc** returns the number of arguments in positions corresponding to and following that of **arg…** as an **int** value. **arg[]** must be supplied an **int** value in the range **0..argc-1**, otherwise it will raise an error condition. It returns the value of the corresponding argument**.**

```
for i in 0..argc-1 do
    dest[i] = arg[i]
endfor
```

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 opt

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

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

1. The name of the procedure
2. The number of values the procedure will return.
3. The number of arguments the procedure will accept
4. The type 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…** 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. Ambiguous types

While PM is designed to facilitate static analysis, it also permits dynamic polymorphism. A value of ***ambiguous type*** is not necessarily subject to static program analysis and may result from one of the following:

- Use of the contents ('**^**') operator on a value of polymorphic or optional polymorphic type.
- Use of a variable or constant initialised to a value of ambiguous type.
- Use of a return value from a procedure call or operator expression with at least one argument of ambiguous type.
- Use of a variable argument access expression **arg[***i***]** where *i* is not a literal constant Use of any parameter value, variable argument access expression or pass-through call within a procedure body that has not been subject to procedure specialisation.

In any of the above cases, program analysis may determine a non-ambiguous type for the given expression.

viii. Procedure specialisation

Procedure specialisation is an optimisation of procedure call selection that should be performed under the following circumstances:

- No argument to the procedure call has an ambiguous type.

In this case the PM system should construct a specialised implementation of the procedure body for the specific combination of argument types 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.

ix. Run-time procedure selection

In most cases procedure selection may be made during program analysis. However, when a procedure call has one or more arguments with ambiguous types, the call may need to be resolved at runtime.

x. Type constraints between different arguments to a procedure

Unlike some other languages adopting 'multi-method' approaches to procedure selection, PM does not provide a specific mechanism to specify that the type of one argument (or component thereof) should match the type of a second argument. Instead, two general language features are provided to determine if the types of any two values match each other. The first is the **?=** operator, which returns a boolean value indicating if its two arguments have the same concrete type. It is most commonly used with **check** statements.

```
proc table_set_value(&table:table,value) check table.first_element?=value
```

This mechanism is sufficient to cope with the majority of cases where type matching between arguments is needed, including the implementation of generic data structures. A PM implementation should be able to locate failed **check ?=** statements during program analysis, providing ambiguous types are not involved.

For those cases where the type matching should modify procedure selection, the **has_same_type** procedure is available. This procedure returns a **null** its two arguments have different types and **affirm** if the two arguments have the same type.

```
proc _set_table(&table:table,value,flag:affirm) do
     _set_table_from_value(&table,value)
endproc

proc _set_table(&table:table,value,flag:null) check value?=table do
     _set_table_from_table(&table,value)
endproc

proc set_table(&table,value) do
     _set_table(&table,value,has_same_type(table.first_element,value))
endproc

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

Note that **has_same_type** has an ambiguous type only if one or both of its arguments have ambiguous types.

9. Variables, Constants, Assignment and Expressions

i. Variables and constants

PM stored values in ***objects*** which may be ***variables*** or ***constants***. Objects are created and associated with names using a declaration statement:

statement::=
    const *assignlist*
    definition
    assignment
definition::=
    (*name*|'_') { **','** (*name*|'_') } **':='***xexpr*
assignlist::=
    assignment { ',' *assignment* } *subexp*
assignment ::=
    lhs **'='** *expr*
    *lhs***','** *lhs*{ **','** *lhs* } **'='** *call*

For example

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

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

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

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

ii. 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 variable.

```
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:

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

For example:

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

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

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

iii. 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.

Operation	Priority	Equivalent call	Description
`x[a,b…]`	0	`proc [] (x,a,b…)`	Array subscript
`x{a,b…}`	0	`proc {} (x,a,b…)`	Array subscript (based on shape)
`^x`	1	`proc ^ (x)`	Obtain value
`x | y`	2	`proc | (x,y)`	If x is not null valued then ^x else 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 over y`	11	`proc over(x,y)`	Change domain of x to conforming domain y
`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

iv. 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.

v. Subscripts and slices

Subscript expressions have the following syntax:

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

Subscript expressions using square brackets **[]** index an element or slice with reference to the domain of the array value being subscripted.

Subscript expressions using braces **{}** index elements or slices with respect to the shape of the array (the domain of its domain).

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)
```

vi. Parameter declarations

A **param** declaration which simply defines a named constant:

paramdec::=
param *name* '**=**' *xexpr*

For example:

```
param pi = 3.14159265
```

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

10. 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* **case** *xexprlist* **do** *statements* {**case** *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
     case '1','3','5','7','9' do print("Odd")
     case '2','4','6','8' do print("Even")
     otherwise do print("?")
endselect
```

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 confirms that an expression yields a **true** value and raises an error if this is not the case. The **debug** statement executes a block of code only if debugging is enabled.

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

For example:

```
check value /= null

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
```

11. 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** *assignlist*] [**conc**] *loopbody* **endfor**
iter::=
name { **,**' *name* } **in** *exp* { **,**'*exp* }
loopbody::=
do *statements* [**let** *assignlist*]
let *assignlist*

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
```

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 it is possible 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 using let

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

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

Loop-invariant expression are described in a later chapter. The **let** clause defines constant 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 constant objects that are created by the **let** clause in the scope enclosing the **find** statement.

find *iter* [*subexpr*] [**using** *keyargs*] [**conc**] [**do** *statements*] *found* **endfind**
found::=
when *xexpr* **let** *constdef* [*subexpr*] *letdefault* { '**;**' *constdef* [*subexpr*] *letdefault* }
letdefault::=
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
let
      loc_above_threshold = pixel_loc default [-1,-1]
      value above threshold = pixel_val default -1
endfind
```

The right-hand-side expressions following **let** are computed for an arbitrary loop invocation for which the **when** expression evaluates to **true**. Unlike **let** 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 constants.

If a **let** 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:

```
let 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 **let** clause and the default expression may both have subexpressions. These subexpressions are independent of each other:

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

The following let clause would cause an error.

```
let 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 match. For a right hand side value *r* and default value *d*, it must be valid to execute: **x:=***d***; x=***r*

The body of a **find** statement may not contain communicating operations, except within nested parallel **for** or **do** statements.

iv. Task parallelism

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

[**using** *keyargs*] [**with** *statements*] **do** *statements* { **also do** *statements* } **enddo**

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 0..***N-1*
using *opts*
do
    $statements_0$
    select *_index*
        case 0 do $statements_1$
        case 1 do $statements_2$
        …
        case *N-1* **do** $statements_N$
    endselect
endfor

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

12. Processor allocation

v. 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 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.

Each PM task is said to own a set of processors. Depending on the implementation, this may be interpreted in one of two ways:

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.

PM assumes that the number of processors available to the program is fixed by the implementation. When a PM program proceeds, this available processor set is divided among available processors according to the following:

On start-up, the PM program is assumed to be running a single task. This task owns all processors in the set.

When a parallel statement (**for**, **find** or **do**.. **also do**) is encountered. Then processor allocation is governed by comparing the size of its associated domain (which is implicit in **do**..**also do**) to the size of the processor set owned by the currently running task

1. If the number of processors in the processor set is greater than the number of elements in the domain, then the processor set is partitioned over the iteration space using the ***default work sharing procedure***. A child task is created to process each element in the iteration space. Each such task will own a processor set containing at least one (and in at least one case more than one) processor from the set owned by the parent task.

2. If the number of elements in the processor set is less than or equal to the number of elements in the domain, then a distribution of the domain over the processor grid is calculated using the ***default data distribution procedure***. A new task is created for each processor owned by the parent task. Each tasks owns a processor set containing only the processor on which it is running and is responsible for processing a subset of the domain.

The PM definition does not specify how a task owning multiple processors utilises those processors to perform its computations. Two possible implementation approaches are:

1. Every processor in the processor set owned by a given task redundantly executes the same code.

2. A single processor in the processor set executes the code, the remaining processors are held in a standby state until needed by nested parallel operations.

There is clearly a trade-off between ease of implementation, inter-processor message overhead and resource utilisation between these two approaches. A PM implementation is not required to employ either of these extremes. The only requirement, as with defining processors, is that controlling implementation overheads should not require intervention from the programmer.

vi. 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 invocations among available processors. Moreover, it set the number of available processors within the statement body to one – thus all dynamically nested for statements are forced to be concurrent.

vii. The using clause

The above process may be modified by providing keyword arguments in the using clause of a **for** or **do** statement. The following options are defined.

distr=*DistributionGenerator*	Use the given distribution in place of the output of the default distribution procedure.
work=*Array*	If this option is specified, the default processor distribution will unevenly distribute processors according to a measure of the relative amount of work associated with each invocation. The given array should have numeric elements and a domain shape equal to the common shape of the inputs to the parallel statement.
block=*Tuple*	This option sets the block size used by the default distribution procedure.

viii. Work sharing

If the number of processors is greater than the size of the domain then processors are allocated to domain elements using a ***worksharing*** procedure.

1. The task processing element of the domain ($\mathbf{x}_j$, *j=1..M*, in domain iteration order) has at least one processor allocated to it, leaving *N* remaining processors (labelled *1..N*) to allocate to tasks.
2. If the **work=***A* option is present in **using**, then processor *i* is allocated to the task processing domain element $\mathbf{x}_j$ where:
   $$\sum_{k=1}^{j-1} A(x_k) < \frac{i}{N+1} \sum_{k=1}^{M} A(x_k) \leq \sum_{k=1}^{j} A(x_k)$$
3. 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*.

ix. Distributions

A ***distribution*** of shape domain *S* associates a subdomain of *S* with each processor in a numbered set 1..*N*. Sub-domains may be described using any iterable value (sequence, tuple of sequences or domain). Distributions also incorporate a ***topology*** which describes a nominal layout of processors which may be used by the implementation to optimise communications. A ***Cartesian gridded topology*** is defined using a tuple of long integer values.

x. Block distribution

A ***block distribution*** divides an *N*-dimensional shape domain by partitioning each dimension *i* into M_i tiles. The shape is divided into $M=M_1 \times \ldots \times M_N$ tiles in total and the topology is defined as $[M_1, \ldots, M_N]$. If a given dimension has N_i elements, then each tile *j*, *j=1..*M_i is defined as *floor((j-1)**N_i*/*M_i*)..floor(j*(*N_i*-1)/*M_i*)*.

A block distribution value is defined using the **block** procedure:

```
distr:= block( grid(1..10,1..8) , [ 3l, 3l] )
```

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

xi. Block cyclic distribution

A ***block cyclic distribution*** divides an *N*-dimensional shape domain by partitioning each dimension into tiles of size B_i. If the topology of the distribution is defined as [$M_1, \dots, M_N$] then the tile associated with processor [$m_1, \dots, m_N$] is $0 \le m_i \le M_i$ defined by the tuple of block sequences [**block_seq(***B_1,m_1, M_1,N1),…,* **block_seq(***B_N,m_N, M_N,N_N)*].

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

```
distr := block_cyclic( grid(1..10,1..8), [2,2], [5,4] )
```

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

xii. Block Sequences

A block sequence defines special type of sequence. Values are created using the **block_seq** function. Values in a block sequence **block_seq(***p,m,M,N***)** are given by:

mp, mp+1, .. , mp+p-1,
mp+Mp, mp+Mp+1, mp+Mp+2, .. , mp+Mp+p-1,
mp+2Mp, mp+2Mp+1, mp+2Mp+2, .. , mp+2Mp+p-1,
.. ,
mp+nMp, mp+nMp+1, mp+nMp+2, .. , min(N,mp+nMp+p-1)

where *n=floor(N/(Mp)).*

This can also be explained by the following code equivalence:

```
for each i in block_seq(p,m,N,N) do
     print(i)
endfor

! is equivalent to

n:=N/(M*p)
for each i in m*p..n*m*p by M*p do
     for each j in 0..p-1 while i+j<N do
          print(i+j)
     endfor
endfor
```

Block sequence types are defined as:

block_seq{ t: range_base }

xiii. Distribution selection

For a **grid** domain the default distribution is **block** unless **block=** is present in **using**, in which case the default distribution is **block_cyclic**.

For a **vector** or **matrix** domain, the default distribution is **block cyclic**.

The default block size for a block cyclic distribution is implementation dependent, but is available through the system-defined parameters: **default_block_size1d** … **default_block_size7d.** This default is overridden if **block**= is present in **using**.

The default processor topology is based on factoring the number of currently available processors to give the same number of processors along each dimension (subject to the prime factors of the number of available processors).

13. Communicating operations

i. Overview

The **for** statement enables each invocation of its enclosed body of statements to be executed concurrently. Numerical models will usually require interaction between invocations 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 a given invocation as if it were an array defined across all invocations, each element corresponding to the local value in a single invocation. This is primarily achieved using '**@**' operators. Used as a unary operator (**@**x) this provides an array view of a given expression across all invocations. Subscripting the result of a unary '**@**' operation, **@**x**[***subs***]**, is treated as an operation in its own right (enabling a more efficient implementation). Used as a binary operator x**@[***ndb_desc***]** returns values from the local neighbourhood of the calling invocation, as defined by the controlling domain.

name '::' *name* ['**@**' *slice*]
name '::' '**(**' [*namelist* ['**;**' arglist]] '**)**' ['**@**' *slice*]
name '%' '**(**' [*namelist* ['**;**' arglist]] '**)**

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. More formally 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.

iii. Channel variables

Communicating operations provide a view of a given value across all invocations of the enclosing for statement. These values must be stored in channel variable. A channel variable is any variable defined within the **for** statement, but not within any **if** or **select** statement nested within the **for** statement (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. Global view operator: unary @

'**@**'*name* '**@**' *name slice*

Used as a unary operator **@x** provides an array view of the value in a channel variable across all invocations of the enclosing **for** statement. The domain of the returned array is equal to the controlling domain of the enclosing for statement. For example, the following code would not yield an error:

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

v. Neighborhood view operator: binary @

name '**@**' *slice*

Used as a subscripting operator **x@[***ndb_desc***]** or **x@{***ndb_desc***}** returns values from a local neighbourhood of the calling invocation. For example:

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

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 a binary **@** 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]
let
      filtered_image=@filtered_pixel
endfor
```

vi. Communicating procedures

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

It is possible to define ***communicating procedures*** incorporating communicating operators or procedures. Such procedures can only be called inside **for** statements:

```
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 invocation use the '**%**' notation. Procedures that return the same value to every invocation use the '**::**' notation.

Arguments to communicating procedures fall into two categories, depending on whether they are positioned before or after the semicolon in the parameter list. Parameters listed before the semicolon are *channel variable* parameters. When the procedure is called, these parameters must be associated with the name of a channel variable. Channel parameters may be used by '**@**' operators and passed as channel arguments to other communicating procedures with regular arguments to a communicating procedure may not. They are additionally associated with synchronisation rules defined below. This restriction does not apply to parameters positioned after the semicolon, which may be associated with general expressions and are not subject to synchronisation rules.

It is possible to require a non-channel 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.

vii. Implicit arguments to communicating procedures

largs::=
['**this_dom**' ['**:**' *type*] '**,**'] ['**this_distr**' ['**:**' *type*] '**,**'] ['**this_tile**' ['**:**' *type*] '**,**'] ['**this_index**' ['**:**' *type*] '**,**']

Communicating procedures are automatically passed four additional arguments: **this_dom**, **this_distr**, **this_tile** and **this_index**, indicating the domain, distribution and tile of the enclosing parallel statement and the index of the current element with respect to the shape of the domain. These four 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 **any**).

```
proc block_comm%(this_distr:block,x)=…
y:=block_comm%(x)
```

viii. 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.

ix. 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 do**..**endproc** form of the procedure body. Each argument to a **local** procedure is passed as an array over the shape of the subdomain being processed on the current processor node (**invar** parameters are passed unmodified.) Local '**%**' procedures must return arrays defined over the same domain as their arguments. Local '**::**' procedures simply return values, however the results are invalid (with undefined consequences) if the same value is not returned on all processor nodes.

x. Synchronisation requirements – structured parallelism

The channel variables associated with PM parallel communication operators must be operated on in the same sequence for each invocation of the enclosing parallel statement. The communication sequence along each possible path through the body of a **for** statement must be the same. This sequence is defined as follows:

1. The sequence associated with a communication operator is defined to be the variable to which the operator is applied (in terms of the object accessed – not just the variable name)
2. The sequence associated with a call to a loop procedure is defined to be the channel variables associated with the call
3. The sequence associated with a **sync** statement is defined to be the variables listed by the statment
4. The sequence associated with a statement list is defined to be an ordered set of the sequences associated with each statement in the statement list which contains communicating operators
5. For a conditional statement (**if**, **select**) the sequences associated with each branch of the conditional must be the same.
6. For a sequential loop (**while**, **repeat**, **for each**) the sequence is defined as the sequence of the loop body, grouped as a ***repeated*** sequence. The form of loop is not relevant - all sequential loops apply the same repeated grouping. In terms of sequence matching, a repeated group must match another repeated group containing the same sequence.

In terms of matching sequences, communicating operators or procedures may only be matched with **sync** statements, not with each other. The **sync** statement supplies the required values to the communicating procedure to which it is matched.

If a conditional expression (**if**, **select**) contains communicating operations then each conditional branch of the statement must contain the same sequence of operations.

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 invocation of the enclosing parallel **for**. If a communicating operation executed in a longer-running loop tries to access a value from an invocation where the loop has already terminated, the operation will be supplied with the value that the named variable possessed immediately prior to the termination of the shorter-running loop. There is an implicit synchronisation at the end of sequential loop.

If a sequential loop containing communicating operations occurs in a conditional statement, then a loop containing a matching sequence of communicating operators must occur at the same position (in terms of sequence of communicating operations) on each 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.

14. Syntax

module::=
    program_module | *library_module*
program_module::=
    [*decls*] *statements*
library_module::=
    decls [**debug** *statements* **enddebug**]
decls::=
    { *import* '**;**' } (*import*| *decl*) { '**;**' *decl* }
import::=
    use *modname* [*modifiers*] { '**,**' *modname* [*modifiers*] }
modname::=
    name { '**.**' *name* }
modifiers::=
    '**{**' *modifier* { '**,**' *modifier* } '**}**'
modifier::=
    [**type** | **param** | **proc**] *name* '**=>**' *name*
decl::=
    procdecl | *typedecl* | *paramdecl*
procdecl::=
    proc *procname* ['**%**'|'**::**'] *params* [*special*] '**=**' *xexprlist* [**do** *statements* **endproc**]
    proc *procname* ['**%**'|'**::**'] *params* [*special*] [proccheck] **do** *statements* [**result** '**=**'*xexprlist*] **endproc**
special::=
    local | **reduce** '**(**' *namelist* '**)**'
procname::=
    name | *opname* | **null** | **opt**
opname::=
    binary | '**^**' |**not** | '**[]**' | '**{}**' | '**[]=**' | '**{}=**' | '**@**'| '**@[]**'| '**@[]||**' | '**@[]**'| '**@[]|**'
    '**@{}**'| '**@{}|**' | '**@_[]**' | '**@_{}**'
binary::=
    '**+**' | '**-**' |'*****' | '**/**' | '******'| '**==**' | '**/=**' | '**>**'| '**>=**' | **mod** | **and** | **or** | '**//**' | '**=>**' | **dim** | **over** | '**..**' | **by**
proccheck::=
    check *exprlist* [*whereclause*]
params::=
    '**(**' largs [cpars ';' | ';'] [*pars* ['**,**' *keypars*] | *keypars*] '**)**'
largs::=
    ['**this_dom**' ['**:**' *type*] '**,**'] ['**this_distr**' ['**:**' *type*] '**,**'] ['**this_tile**' ['**:**' *type*] '**,**'] ['**this_index**' ['**:**' *type*] '**,**']
cpars::=
    name['**:**' *type*] { '**,**' *name*['**:**' *type*] }
pars::=
    { *param* '**,**' } (*param* | **arg** '**...**')
param::=
    ['**&**'] *name*['**:**' [**invar**] *type*]
keypars::=
    { *name* '**=**' *expr* '**,**' } (*name* '**=**' *expr* | **key** '**...**')
namelist::=
    name { '**,**' *name* }
typedecl::=
    type *name* ['**{**' *typeparams* '**}**'][**in** *namelist*] [**is** *typelist* | **includes** *typelist* | **also includes** *typelist*]
typeparams::=
    name '**:**' *type* { '**,**' *name* '**:**' *type* }
paramdecl::=
    param *name* '**=**' *xexpr*
typelist::=

type { '**,**' *type* }

type::=
any | **null** | **affirm**
name ['**{**' *opttypelist* '**}**']
[*type*] '**[**' *opttypelist* '**]**'
'**<**' *type* '**>**'
'**<**' **opt** *type* '**>**'
[**struct** | **rec**] [*name*] '**{**'*name* ['**:**' *type*] { '**,**' *name* ['**:**' *type*] } '**}**'
opttypelist::=
[*type*] { '**,**' [*type*] }
*statements***::=**
statement { '**;**' *statement* } ['**;**']
*statement***::=**
if *xexpr* **then** *statements* { **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**
select *xexpr* { **case** *xexprlist* **do** *statements* } [**otherwise** *statements*] **endselect**
while *xexpr* **do** *statements* **endwhile**
repeat *statements* **until** *xexpr*
for each *iter subexpr* [**until** *xexpr*] **do** *statements* **endfor**
sync '**(**' [*namelist*] '**)**'
check *expr* [*whereclause*]
debug *statements* **enddebug**
const *constdefs* [*subexpr*]
for *iter* [*subexpr*] [**using** *keyargs*] [**conc** | **accel**] *forbody* **endfor**
find *iter* [*subexpr*] [**using** *keyargs*] [**conc** | **accel**] [**do** *statements*] *found* **endfind**
[**using** *keyargs*] [**with** *statements*] **do** *statements* [*letclause*] { **also do** *statements* [*letclause*] } **enddo**
assignment
vardef
constdef
call
iter::=
name **in** *expr*
name '**,**' *iter* '**,**' *expr*
forbody::=
letclause | **do** *statements* [*letclause*]
letclause::=
let *constdef* [*subexpr*] { '**;**' *constdef* [*subexpr*] }
found::=
when *xexpr* **let** *constdef* [*subexpr*] *letdefault* { '**;**' *constdef* [*subexpr*] *letdefault* }
letdefault::=
default ('**(**' *exprlist* '**)**' | *expr* | *call*) [*subexpr*]
deflist::=
definition { ';' *definition* }
definition::=
name '**:=**' *xexpr*
deflhs '**,**' *deflhs* { '**,**' *deflhs* } '**:=**' *call*
deflhs::=
name | '**_**'
assignlist::=
assignment { ',' *assignment* } *subexp*
assignment ::=
lhs '**=**' *expr*
lhs'**,**' *lhs* { '**,**' *lhs* } '**=**' *call*
lhs::=
ref | '**_**'

ref::=
name
ref qualifier
'**^**' *name*
'**^**' '**(**' *ref* '**)**'
constdefs:=
constdef { '**,**' *constdef* }
constdef::=
name '**=**' *expr*
deflhs '**,**' *deflhs* { '**,**' *deflhs* } '**=**' *call*
xexpr::=
expr subexp
subexpr::=
[**check** *exprlist*] *whereclause*
whereclause::=
{ **where** *constdefs* }
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 **over** *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*
'**^**' *expr*
term qualifier
term::=
name
value
'**(**'*expr* '**)**'
'**[**' *exprlist* '**]**'
call
array
'**<**' [**opt**] [*type*] '**>**' *term*
[**struct** | **rec**] [*name*] '**{**' *name* '**=**'*exp* { '**;**' *name* '**=**'*exp* } '**}**'
'**@**' *name* [*slice*]
name '**@**' *slice*

value::=
arg '**[**'*expr*'**]**' | **argc** | **true** | **false** | **null** | **affirm**
proc *procname*
number | *string*
call::=
(*name* | **opt** | **null** | **proc** *procname*) '**(**' *arglist* '**)**'
(*name* | **proc** *procname*) '**%**' '**(**' *parglist* '**)**'
(*name* | **proc** *procname*) '**::**' '**(**' *parglist* '**)**' ['**@**' *slice*]
(*name* | **proc** *procname*) '**::**' *name* ['**@**' *slice*]
parglist ::=
namelist
[*namelist* '**;**'] arglist
arglist::=
{ *arg* '**,**' } (*arg* | **arg** '**...**') ['**,**' *keyargs*]
[*keyargs*]
arg::=
'**&** *ref* | *expr*
keyargs::=
name '**=**' *expr* { '**,**' *name* '**=**' *expr* }
qualifier::=
{ '**.**' *name* | *slice* }
slice ::=
'**[**' [*expr*] { '**,**' [*expr*] } '**]**'
'**{**' [*expr*] { '**,**' [*expr*] } '**}**'
array::=
'**(**' *list2d* '**)**'
'**{**' *list2d* '**}**'
list2d::=
exprlist {'**;**' *exprlist*} ['**;**']

15. 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

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
`max(x:any_real,y:any_real)`	Maximum
`min(x:any_real,y:any_real)`	Minimum

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.

`balance(x:num,y:num)`	Numerical balancing

The balance procedure returns *x* 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,y)`	Format value as string
`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

`?=(x,y)`	Type comparison

1. The result type of this procedure is **bool**.
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).

`has_same_type(x,y)`	Type comparison

1. The result type of this procedure is either **null** or **affirm**.
2. The result is **affirm** 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 **null**.

ix. Range and sequence creation

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.

x. Range and sequence information

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

xi. Sequence manipulation

`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*

xii. Domain creation

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

xiii. Domain information and manipulation

`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(*x*)**	index
`conform(x:dom,y:dom)`	Domain y conforms to domain x	bool
`size(x:dom)`	Number of elements in domain	long

xiv. Distribution creation

`block(dom:grid,topol:tuple)`	Create block partition
`block_cyclic(dom:grid,block:tuple,topol:tuple)`	Create block cyclic partition

All arguments must be values with the same number of dimensions

xv. Processor grouping

`sys_prc()`	Processor rank within the set of all available processors
`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_prc()`	Processor rank within set of processors owned by current task
`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.

xvi. Communicating operations

`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