Mesham Language Specification

Version 1.0a_2

October 2012

Contents

1 Introduction

Mesham is a programming language designed to simplify High Performance Computing (HPC) yet result in highly efficient executables. This is achieved mainly via the type system, the language allowing for programmers to provide extra typing information not only allows the compiler to perform far more optimisation than traditionally, but it also enables conceptually simple programs to be written. The language is designed such that to support relatively simple, efficient, portable and safe code.

This document contains the version one language specification for both programmers and compiler implementors. The specification is independant from target platform and it is the job of the language implementor to realise the specification on their chosen architecture.

The reader should consider the execution of a Mesham program within the confines of an abstract parallel machine as defined in this document. Broadly, this machine is divided into a number of processing units, with one unit per parallel process. The machine as a whole has, as far as the specification is concerned, an infinite amount of memory which is shared between the processing units. Along with the memory is also a set of environments which bind program symbols to their corresponding storage locations. Each processing unit has some form of connection to every other unit, along which messages may be sent.

1.1 Meta Characters

In order to explain many of the syntactic aspects of the language, meta characters will be used. These are detailed in table 1 and will be found throughout the specification.

Characters	Description
{}	Optional
{}*	Zero or more
{}+	One or more
name	A variable name
...	Continuation

Table 1: Meta Characters used in the specification

2 Type oriented programming

Much work has been done investigating programming paradigms. Common paradigms include imperative, functional, object oriented and aspect oriented. However, we have developed the idea of type oriented programming. Taking the familiar concept of a type we have associated in depth runtime semantics with such, so that the behaviour of variable usage (i.e. access and assignment) can be determined by analysing the specific type. In many languages there is the requirement to combine a number of attributes with a variable, to this end we allow for the programmer to combine types together to form a supertype (type chain.)

2.1 Type chains

A type chain is a collection of types, combined together by the programmer. It is this type chain that will determine the behaviour of a specific variable. precedence in the type chain is from right to left (i.e. the last added type will override behaviour of previously added types.) This precedence allows for the programmer to add additonal information, either perminantly or for a specific expression, as the code progresses. Figure 1 illustrates a type chain and the precedence of such.

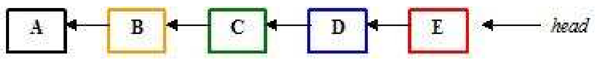

Figure 1: Illustration of type chain precedence

3 Underlying concepts

3.1 The environment

The environment binds program elements to their storage location, the semantics of the specific element defines exactly where that binding occurs. Memory, which is used to store these elements, comes from three distinct areas. The specific area used depends upon the program context and type information.

Stack

Each function operates within its own private stack frame which is alive for the duration of that single function call. Dependant upon the limits of the machine, the size of this frame has no upper limit and can grow as storage requirements dictate.

Heap

Mesham provides for a memory heap which exists regardless of the current execution state. Elements may be stored onto the heap, although unlike the stack which automatically frees the allocated memory upon function exit, the heap does not operate in this fashion. This language specification does not define how one should maintain heap consistency and free no longer used memory. Multiple options are available to the language implementor such as automatic freeing when the variable goes out of scope and garbage collection.

Static

This memory is allocated during compile time; constants and identifiers allocated here have a lifetime extending across the entire run of the program. Program variables allocated in static memory will be initalised once only. All constant strings are held within static memory.

3.2 Process interconnect

Shared memory

Mesham implements the Logic Of Global Synchrony (LOGS) model of shared memory communication by default. Using this model the programmer can synchronise on a single variable, group of variables or globally. One can consider a vector, *w*, of all program variables (global synchronisation), subset of variables (group synchronisation) or a single variable (single variable synchronisation) w such as $< \overleftarrow{w}, w_0, w_1, w_2, \dots, i_n, \vec{w} >$. This specifies that w starts in initial state $\overleftarrow{w}$ (pre-w) and after n intermediate steps, if the program terminates, then the final state of w is $\vec{w}$ (post-w.) Each w_k (where k<n) denotes the state at the *k-th* internal synchronisation point, hence w_0 is the state after the first synchronisation of *w*. Such a denotation is termed *feasible* if, for any beginning state, there exists some final state and a set of intermediate states. To an outside, global observer, there is no guarantee of the state during the intermediate steps, only the pre-w ($\overleftarrow{w}$) and post-w ($\vec{w}$) states are guaranteed. Axiomatically, one can consider $\overleftarrow{w}$ as the precondition and $\vec{w}$ the post condition.

This should be implemented in a manner which guarantees safety and consistency. The programmer must be able to operate within a shared memory context in the knowledge that their code, if it compiles, will function in a manner that is safe although it is accepted that this might incur a performance penalty.

Message passing

The language also supports the message passing style of communication which allows for individual or groups of messages to be sent between individual, groups or all processes within the parallel system. Unlike shared memory, there is no requirement that the message passing implementation guarantees safety or consistency implicitly. It should be up to the programmer to consider safety by, for example, ensuring that messages complete.

4 Lexical Conventions

4.1 Comments

Comments can be indiciated in one of two ways. Either the characters // will denote a single line comment or the matching of /* and */ allow for multi line comments.

4.2 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter with upper and lower case letters considered different. The underscore character may be included and is counted as a letter. Identifiers may be of any length. Identifiers prefixed with MESHAM_ are reserved and may not be used within programmer codes.

4.3 Keywords

The following 26 identifiers are reserved for use as keywords and may not be used otherwise throughout code:

break	catch	continue	currenttype
declaredtype	else	false	for
from	function	group	if
null	par	proc	return
skip	sync	throw	to
true	try	typevar	var
	void	while	

4.4 Constants

There are several kinds of constants each of which has an associated data type

```
constants:
        integer
        character
        floating point
        string literal
        boolean
        null
```

Integer

All integer constants are taken to be decimal base ten.

Character

A sequence of one or more characters enclosed in single quotes e.g. *'a'*. The value of a character constant with only one character is the numeric value representation in the machine's character set. The following escape characters are supported although how they are interpreted is implementation defined:

newline	\n	horizontal tab	\t
vertical tab	\v	backspace	\b
carriage return	\r	formfeed	\f
audiable beep	\a		

Floating point

Consists of an integer part, a decimal point and then the fraction part. The integer aspect may be omitted. Such a value is inferred to be a double precision floating point, the programmer may specify single precision floating point explicitly.

String literal

A string literal, otherwise known as string constant, is a sequence of characters surrounded by double quotes e.g. *"..."*. String literals are unmodifiable and once constructed may not be changed (although can be composed into further strings.)

Boolean

Mesham recognises the *true* and *false* constants which relate to their boolean values.

Null

The null constant represents no value and a variable containing this guarantees not to reference a valid location in memory.

4.5 Code blocks

Blocks of code are represented via the { and } symbols. When concerned with composition precedence, then code blocks acts similar to operator braces, i.e. have the higher precedence.
Mesham uses lexical scoping, where each code block has a specific associated scope and its own environment. Nested blocks can reference the identifiers and environments of higher level blocks in the nest.

4.6 Types

Types are central to the concept of type oriented programming and Mesham. Whilst specific types are defined later in this document, a type is:

```
type =     elementtype
         | compoundtype
         | type :: type
         | type variable name
```

The *elementtype* and *compoundtype* are defined later in the specification.

5 Preprocessor

The preprocessor will run prior to compilation and performs syntactic transformations on the code. All preprocessor directives as prefixed with the # hash character.

5.1 Include

Syntax: #include “filename”

Semantics: Will include the contents of the Mesham file denoted by *filename* into the current file at that point to be included within compilation. This searches the current directory for the specified file.

Alternatively the <name> syntax may be used, which will locate the file called *name* within the Mesham system include directory. This is useful including in-built Mesham files.

6 Imperative Language

6.1 Structure

All control flow must be contained within a function. Both normal and type variables may be defined outside a function and the scope of these is set to be global. The programmer is required to define a program entry point (via the main function.)

6.2 Composition

All code statements and blocks must be terminated by a form of composition. Parallel composition has a higher precedence than sequential composition, such that *A || B ; C* will execute A and B in parallel and then once these have run to completion then C will be executed. All composition is left associative.

Sequential composition

Syntax: a body ; b body

Semantics: Code block *a body* will execute and once it has finished then *b body* will execute.

Parallel composition

Syntax: a body || b body

Semantics: The parallel equivalent of sequential composition, code blocks *a body* and *b body* will execute at the same time.

Example:

```
var j:=23 || {var q:=9; print(q,"\n")};
```

One process will declare *j* to be 23, whilst the other will declare *q* to be 9 and display it.

6.3 Declaration

All variables must be declared before use. Mesham provides two ways of supporting this; declaration through values or through explicit types. It is at variable declaration that the environment shall map the identifier name to storage location. This is done depending upon the type information provided and if no such explicit type is present the infered type is used.

Value based declaration

Syntax: var name{:=value};

Semantics: Will define the variable in the current environment and assign a value to it if provided. All declared but unassigned variables have the value of *null*.

Examples:

```
var a;
var b:=23;
```

Variable *a* is defined, but no value associated (therefore *null* at this point.) Variable *b* is defined to be the value 23 and, by type inference, has type *Int*.

Notes: One is not required to specify the value of the variable at this point, if no value is available then the type will be infered during the initial assignment. It is illegal to declare a variable using *null* as the value to infer the type from. This is because *null* is a special, no value, variable which has no specific type.

Type based declaration

Syntax: var name{:type};

Where type is a *type*, variable name or *type :: type.* The operator *:* sets the type and *::* is type combination (coercion).

Semantics: This will declare a variable to be a specific type. Type combination is subject to a number of semantic rules. If no type information is given, then the type will be found via inference where possible.

Examples:

```
var i:Int :: allocated[multiple[]];
```

Here the variable *i* is declared to be integer, allocated to all processes. There are three types included in this declaration, the element type *Int* and the compound types *allocated* and *multiple.* The type *multiple* is provided as an argument to the allocation type *allocated*, which is then combined with the *Int* type.

```
var m:String;
```

In this example, variable *m* is declared to be of type *String.* For programmer convenience, by default, the language will automatically assume to combine this with *allocated[multiple]* if such allocation type is missing.

6.4 Assignment

Value assignment

Syntax: lvalue:=rvalue; *(where rvalue is a memory reference or expression, lvalue is a memory reference)*

Semantics: rvalue is assigned to lvalue

Examples:

```
var a;
var b:=99;
a:="hello";
```

In this example variable *a* is defined, but no value associated initially. As the program progresses the string *"hello"* is assigned to *a* and by type inference the type of this variable becomes *String.* Variable *b* is defined to be the value 99 and, by type inference, has type *Int.*

Notes: The value assignment must be allowed with respect to the types of the variables and/or constants involved. Some assignments will be illegal and disallowed.

Type assignment

Syntax: name:type;

Semantics: Will modify the type of an already declared variable via the *:* operator. Allocation information may not be changed.

Examples:

```
var i:Int :: allocated[multiple[]];
i:=23;
i:i :: const[];
```

Here the variable *i* is declared to be integer, allocated to all processes and its value is set to 23. Later on in the code the type is modified to set it also to be constant (so from this point on the programmer may not change the variable's value.) In this third line *i:i :: const[];* sets the type of *i* to be that of *i* combined with the *const* type.

Notes: Changing the type will not have any runtime code generation in itself, although the modified semantics will affect how the variable behaves from that point on. Due to the type being a static compilation notion only, each scoped block will limit the typed scope of a variable and have the type associated with it. For example, if one changes the type of a variable then if the variable lives beyond the current block then when that scope is left, the type is reverted back to the previous scope's type. One may not modify the underlying allocation type, once set, during program execution. The environment maps the identifier to a memory location and this storate location may not be modified, although the meaning of such a location can be via coercion.

6.5 Type Support

currentype

Syntax: currentype varname

Semantics: Will return the current type of the variable.

Example:

```
var i: Int;
var q:currentype i;
```

Will declare *q* to be an integer the same type as *i*.

Notes: This is a static construct only and its lifetime is limited to the compilation of Mesham code. As such it must be statically deducable and the language should take steps to stop the programmer using such a construct dynamically.

declaredtype

Syntax: declaredtype varname

Semantics: Will return the declared type of the variable.

Example:

```
var i:Int;
i:i::const[];
i:declaredtype i;
```

Here in line 2 the programmer adds the constant type to the variable, however the type is then reverted back to the declared type (integer) in line 3.

Notes: This is a static construct only and its lifetime is limited to the compilation of Mesham code. As such it must be statically deducable and the language should take steps to stop the programmer using such a construct dynamically.

Type variables

Syntax:
typevar name{::=type};
name::=type;

Note how *::=* is used rather than *:=*, typevar is the type equivalent of var

Semantics: Type variables allow the programmer to assign types and type combinations to variables for use as normal program variables. These exist only in compilation and are not present in the runtime semantics.

Examples:

```
typevar m::=Int :: allocated[multiple[]];
var f:m;
typevar q::=declaredtype f;
q::=m;
```

In the above code example, the type variable *m* has the type value *Int :: allocated[multiple[]]* assigned to it. On lines 2 and 3, new (program) variables are created using this new type variable. In line 4, the type variable *q* is declared and has the value of the declared type of program variable *f.* Lastly in line 5, type variable q changes its value to become that of type variable m. Although type variables can be thought of as the programmer creating new types, they can also be used like program variables in cases such as equality tests and assignment.

Notes: Type variables only exist statically and as such must be determinable during compilation. The language should prevent the programmer from using type variables in a manner which can only be deduced at runtime.

6.6 Imperative Control Flow

Conditional

Syntax:
```
if (condition)
    then body;
{ else
    else body;}
```

Semantics: If the condition is true then execute the *then body*, otherwise execute the *else body* (if it exists.)

While loop

Syntax:
```
while (condition)
    while body;
```

Semantics: Loops whilst the boolean condition holds.

For loop

Syntax:
```
for i from a to b
    for body;
```

Semantics: Increments the loop variant on each iteration, starts from integer a and will continue to loop through whilst a is smaller or equal to b. *Notes:* The loop range (a and b) must be integers and the looping variant and/or the ranges can be modified during iteration if so wished.

Break

Syntax: break;

Semantics: Will break out of the directly enclosing loop.

Continue

Syntax: continue;

Semantics: Will continue execution of the loop at the next iteration, i.e. will ignore the remainder of the current loop iteration.

Skip

Syntax: skip;

Semantics: A no operation, does nothing!

6.7 Parallel Control Flow

Par loop

Syntax:
```
par p from a to b
    par body;
```

Semantics: The parallel equivalent of the for loop, each iteration will execute concurrently on different processes. This allows the programmer to write code MPMD style, with the limitation that bounds a and b must be known during compilation.

Example:

```
var p;
par p from 0 to 9
{
        print("Hello from process ",p,"\n");
};
```

Notes: All (variable sharing) communication in a par loop is performed using one sided communication, whereas variable sharing SPMD style is performed using synchronous communication for performance reasons. The code fragment will spawn 10 processes (0 to 9 inclusive) and each will display a message.

Single process selection

Syntax:
proc n
    proc body;

Semantics: This will limit execution of a block to a certain process, n must be an integer constant or variable.

Example:

```
proc 0
{
        print("Hello from 0\n");
};

proc 1
{
        print("hello from 1\n");
};
```

The code example will run on two processes, the first will display the message *Hello from 0*, whilst the second will output the message *hello from 1.*

Group process selection

Syntax:
group n_1,n_2, …,n_d
    group body;

Semantics: Will limit execution of a block to a certain group of processes, each n must be an integer constant or variable.

Synchronisation

Syntax: sync {name};

Semantics: Will synchronise processes where they are needed. For instance, if using the asynchronous communication type, the programmer can synchronise with a variable name and the keyword will ensure all communications of that variable are up to date. One sided communication (variable sharing MPMD style in a par loop) is also linked into this keyword and it will ensure all communication is completed. Without a variable will synchronise all outstanding variables that need synchronising. If a process has no variables that need syncing then it will ignore this keyword and continue.

6.8 Functions

Calling

Syntax: functionname(arg_1, arg_2, …, arg_n);

Semantics: Will call the specific function with the arguments specified. Mesham is pass by reference for all variable arguments and constants (including the result of an expression.) Constants are read only however and may not be modified from within the called function. Hence all provided expressions to a function should be evaluated prior to the call and the memory holding the result supplied to the function itself. Functions may return a value and as such the function call can be used as an expression in itself.

Only constants, expressions and variables may be provided to functions, other identifiers are illegal within this context.

Specification

Syntax: function returntype name(arguments)

Semantics: Specifies a function but does not provide a body. This is useful when one wishes to indicate that a function is available but it might not be desired to include the entire body and associated code. The requirement is that associated function body must be available at runtime.

Notes: One may call native code written in other languages via this mechanism as long as the native function call is visible during program linkage and/or at runtime.

Functions may only be defined at the top level within a program, the concept of nested function is not supported by Mesham.

Declaration

Syntax: function returntype name(arguments)
  function body

Semantics: Declared a function and its body.

Example:

```
function Int add(var a:Int,var b:Int)
{
        return a + b;
};
```

This function takes two integers and will return their sum. All functions operate within their own stack frame, which is unique on a call by call basis. This stack frame can hold function housekeeping specific items (such as the return address) and, depending upon type information, is where variables are allocated to.

Main function

Returns void and can have either 0 or 2 arguments. If present the first argument is number of command line interface parameters passed in and the second is a String array containing these. Location zero of the string array is the program name.

Return

Syntax: return value

Semantics: Returns either value or memory reference and control flow from the function back to the caller.

Notes: The value returned must be compatible with the return type in the function definition.

Returning either a stack frame allocated variable or a constant from a function will return the value. Returning either a heap or statically allocated variable will return a reference to the memory location.

6.9 Error Handling

try

Syntax:
try
    try body
catch (error string)
    error handing code;

Semantics: Will execute the code in the try body and handle any errors.

Notes: The error string can either be one of the in build Mesham error strings or alternatively a programmer defined error string.

throw

Syntax: throw error string;

Semantics: Will throw the error string, and either cause termination of the program or, if caught by a try catch block, will be dealt with.

Example:

```
try
{
        throw "an error"
} catch "an error" {
        print("Error occurred!\n");
};
```

In this example, a programmer defined error (*an error*) is thrown and caught.

In built error strings

Table 2 details the in build error strings supported by Mesham and maybe thrown during runtime. It is possible for the programmer to define their own additional error strings.

String	Description
“”	All errors
“Array Bounds”	Accessing an array outside its bounds
“Divide by zero”	Divide by zero error
“Memory Out”	Memory allocation failure
“root”	Illegal root process in communication
“rank”	Illegal rank in communication
“buffer”	Illegal buffer in communication
“count”	Count wrong in communication
“type”	Communication type error
“comm”	Communication communicator error
“truncate”	Truncation error in communication
“Group”	Illegal group in communication
“op”	Illegal operation for communication
“arg”	Arguments used for communication incorrect

Table 2: Error strings supported by Mesham

6.10 Expressions

Mesham supports a variety of expressions which can be used in a variety of ways such as the *rvalue* in assignments, conditionals and self modification. An expression is made up of:

```
expression:
        expression operator expression ,
        expression operator ,
        operator expression ,
        variable ,
        constant ,
        function call
```

Some combinations of expressions (such as addition with string literals and itegers) are undefined and hence illegal.

Supported operators

Mesham supports a variety of operators, these are detailed in table 3.

Operator	Description	Precedence level	Associativity
( )	Function call or expression parentheses	1	left to right
[ ]	Array element access	1	left to right
#	Array element access	1	left to right
.	Member access	1	left to right
.	Member access	1	left to right
++expr	Pre fix increment	2	right to left
–expr	Pre fix decrement	2	right to left
+ -	Unary plus or minus	2	right to left
!	Logical negation	2	right to left
* /	Multiplication, division or modulus	3	left to right
+ -	Addition or subtraction	4	left to right
< <=	Relational less than/less than or equal to	5	left to right
> >=	Relational greater than/greater than or equal to	5	left to right
== !=	Relational is equal to/is not equal to	6	left to right
&&	Logical AND	7	left to right
‖	Logical OR	8	left to right
?:	Ternary conditional	9	right to left
:=	Assignment	10	right to left
+= -=	Addition/subtraction assignment	10	right to left
expr++	Post fix increment	11	left to right
expr–	Post fix decrement	11	left to right

Table 3: Operators supported by Mesham

7 Type Library

Broadly the type library is split into *element types* and *compound types*. Element types are used to specify the type of a specific data element whereas compound types are used to specify the type of multiple elements or to provide additional information on how to handle a specific variable. All type chains must contain an element type at some point (either directly or embedded within a compound type.) Optionally types may have arguments associated with them which can be any constant, identifier and/or type chain depending upon the type. Syntactically, this information is provided via [ ] which may be omitted.

Types with a capital first letter are designated element types and define the atomic type of the chain. All type chains must have atleast one element type and whilst "the view" of the type can be coerced during program flow, the underlying data allocation element type can not be modified once set.

7.1 Attribute

Const

Syntax: const[ ]

Semantics: Enforces the read only property of a variable.

Example:

```
var a:Int;
a:=34;
a:(a :: const[]);
a:=33;
```

The code in the above example will produce an error. Whilst the first assignment (*a:=34*) is legal, on the subsequent line the programmer has modified the type of *a* to be that of *a* combined with the type *const*. The second assignment is attempting the modify a now read only variable and will fail.

Tempmem

Syntax: tempmem[ ]

Semantics: Used to inform the compiler that the programmer is happy that a call (usually communication) will use temporary memory. Some calls can not function without this and will give an error, others will work more efficiently with temporary memory but can operate without at a performance cost. This type is provided because often memory is at a premium, with applications running towards at their limit. It is therefore useful for the programmer to indicate whether or not using extra, temporary, memory is allowed.

Share

Syntax: share[name]

Semantics: This type allows the programmer to have two variables sharing the same memory (the variable that the share type is applied to uses the memory of that specified as arguments to the type.) This is very useful in HPC applications as often processes are running at the limit of their resources. The type will share memory with that of the variable *name* in the above syntax. In order to keep this type safe, the sharee must be smaller than or of equal size to the memory chunk, this is error checked.
Example:

```
var a:Int::allocated[multiple[]];
var c:Int::allocated[multiple[] :: share[a]];
var e:array[Int,10]::allocated[single[on[1]]];
var u:array[Char,12]::allocated[single[on[1]] :: share[e]];
```

In the example above, the variables a and c will share the same memory. The variables e and u will also share the same memory. There is some potential concern that this might result in an error - as the size of u array is 12, and size of e array is only 10. If the two arrays have different types then this size will be checked dynamically - as an int in C is usually 32 bit and a char usually only 8 then most likely this sharing of data would work in this case.

Extern

Syntax: extern[location]

Semantics: Provided as additional allocation type information, this tells the compiler NOT to allocate memory for the variable as this has been already done externally. The *location* argument is optional and just tells the compiler where the variable is to be found (e.g. a C header file) if required.

Directref

Syntax: directref[ ]

Semantics: This tells the compiler that the programmer might use this variable outside of the language (e.g. Via embedded C code) and not to perform certain optimisations which might not allow for this.

Example:

```
var pid:Int :: allocated[multiple[]] :: directref[];
```

7.2 Allocation

There are a number of types which the programmer can use to specify how and where a variable is located within the memory of different processes. Just this task alone adds many keywords to existing parallel languages which, using the proposed type approach, is avoided.

Allocated

Syntax: allocated[type];

Semantics: This type sets the memory allocation of a variable, which may not be modified once set.

Example:

```
var i: Int :: allocated[];
```

In this example the variable *i* is an integer. Although the *allocated* type is provided, no addition information is given and as such Mesham allocates it to each processor.

Default: In the absence of further information this will default to include the *multiple* type thus allocating the variable amongst multiple nodes.

Multiple

Syntax: multiple[type];

Semantics: Included in allocated will (with no arguments) set the specific variable to have memory allocated to all processes within current scope.

Example:

```
var i: Int :: allocated[multiple[]];
```

In this example the variable *i* is an integer, allocated to all processes.

Default: By default assumes no type and as such allocated to multiple nodes.

Commgroup

Syntax: commgroup[process list]

Semantics: Specified within the multiple type, will limit memory allocation (and variable communication) to the processes within the list given in this type's arguments.

Example:

```
var i:Int :: allocated[multiple[commgroup[1,2]]];
```

In this example there are a number processes, but only 1 and 2 have variable *i* allocated to them.

Single

Syntax: single[type]
single[on[process]]

Semantics: Will allocate a variable to a specific process. Most commonly combined with the *on* type which specifies the process to allocated to, but not required if this can be inferred. Additionally the programmer will place a distribution type within *single* if dealing with distributed arrays.

Example:

```
var i:Int :: allocated[single[on[1]]];
```

In this example variable *i* is declared as an integer and allocated on process 1.

Stack

Syntax: stack[]
Semantics: Instructs the environment to bind the associated variable to stack frame memory, which exists in runtime only whilst a specific function call is operative. Once the function has returned from its call then this memory is freed and future calls to the same function reference different stack frame memory.

Example:

```
var i:Int :: allocated[stack[]];
```

In this example variable *i* is declared as an integer and allocated onto the stack frame of the currently executing function.

Heap

Syntax: heap[]
Semantics: Instructs the environment to bind the associated variable to heap memory.

Example:

```
var i:Int :: allocated[heap[]];
```

In this example variable *i* is declared as an integer and allocated onto the heap.

Static

Syntax: static[]
Semantics: Instructs the environment to bind the associated variable to static memory. This binding occurs only once for a specific unique identifier with all future declarations sharing the same physical location irrespective of the lexical binding rules.

Example:

```
var j;
for j from 0 to 9 {
var i:Int :: allocated[static[]];
};
```

In this example loop, variable *i* is declared as an integer and allocated into static memory. Because it is allocated into static memory the physical memory used is the same per loop iteration and environment binding occurs only once.

7.3 Element Types

An element type is a primitive type and allocates the associated identfier onto the stack frame of the currently executing function by default. Mesham supports a number of element types, these are detailed in table 4.

Communication in Assignment: When a variable is assigned to another, depending on where each variable is allocated to, there may be communication required to achieve this assignment. Table 5 details the communication rules in the assignment *assignmed variable := assigning variable.* If the communication is issued from MPMD programming style then this will be one sided. The default communication listed here is guaranteed to be safe, which may result in a small performance hit.

Example:

```
var a:Int;
var b:Int :: allocated[single[on[2]]];
var p;
par p from 0 to 3
{
        if (p==2) {b:=p};
        a:=b;
};
```

This code will result in a onesided broadcast (due to being written MPMD style in *par* loop) where process 2 will broadcast its value of b to all other processes who will write it into a. As already noted, in absence of allocation information the default of allocating to all processes is used. In this example the variable *a* can be assumed to additionally have the type *allocated[multiple].*

Element Subtypes

In this document the subtype notation A $\preceq$ B is used to signify that A is a subtype of B. Subtypes are transitive and antisymmetric. Implicit coercions exist both ways between subtypes, although there is no guarantee that the data or precision will be maintained (in fact in some situations there is no way it can be.) Note that these subtypes are purely conceptual and do not specify how the compiler designer should implement the language.

$$\text{Bool} \preceq \text{Char} \preceq \text{Short} \preceq \text{Int} \preceq \text{Float} \preceq \text{Long} \preceq \text{Double}$$

7.4 Collection

Collections are a number of elements types associated together in some sort of structure or abstraction. By default collection types force their associated variables to be allocated to the heap.

Array

Syntax: array[type,d_1,d_2,...,d_n]

Semantics: An array, where *type* is the element type, followed by the dimensions. The programmer can provide any number of dimensions to create an n dimension array. Default is row major allocation (although this can be overridden via types.) In order to access an element of an array, the programmer can either use the traditional *name*[*index*] syntax or, alternatively *name#index* which is preferred by the thesis author.

Type	Description
Int	Integer (32 bit)
Short	Short integer (16 bit)
Float	Floating point number (32 bit)
Double	Double precision number (64 bit)
Long	A long (64 bit) integer
Char	A character (8 bit)
Bool	True or false value
String	A string of characters
File	A file handle

Table 4: Mesham's element types

Assigned Variable	Assigning Variable	Semantics
multiple[ ]	multiple[ ]	local assignment
single[on[i]]	multiple[ ]	local assignment on process i only
multiple[ ]	single[on[i]]	MPI broadcast from process i
single[on[i]]	single[on[i]]	local assignment on process i
single[on[i]]	single[on[j]]	sent from j and received by i ($i \neq j$)

Table 5: Element type communication in assignment

Communication of Assignment: When an array variable is assigned to another, depending on where each variable is allocated to, there may be communication to achieve this assignment. Table 6 details the communication rules for this assignment *assignmed variable := assigning variable.* As with the element type, default communication of arrays is safe.

Example:

Assigned Variable	Assigning Variable	Semantics
multiple[ ]	multiple[ ]	memory copy
single[on[i]]	multiple[ ]	memory on process i only
multiple[ ]	single[on[i]]	MPI broadcast from process i
single[on[i]]	single[on[i]]	local memory copy on process i
single[on[i]]	single[on[j]]	sent from j and received by i ($i \neq j$)

Table 6: Array type communication in assignment

```
var a:array[String,2] :: allocated[multiple[]];
(a#0):="Hello";
(a#1):="World";
print((a#0)," ",(a#1),"\n");
```

This example will declare variable a to be an array of 2 Strings. Then the first location in the array will be set to *"Hello"* and the second location set to *"World"*. Lastly the code will display on stdio both these array string locations followed by newline.

Default: In the absence of additional type information an array will be allocated in a column major fashion.

Row and Col Types

Syntax: row[ ]
col[ ]

Semantics: In combination with the array, the programmer can specify whether allocation is row or column major. This allocation information is provided in the allocation type. Formally, and applicable to larger dimensions, in row major allocation the first dimension is the most major and the last most minor. In column major allocation the first dimension is minor and the last is most major.

Example:

```
var a:array[Int,10,20] :: allocated[col[] :: multiple[]];
((a#1)#2):=23;
(((a :: row[]) #1)#2):=23;
```

Where the array is column major allocation, but the programmer has overridden this (just for the assignment) in line 3. If one array of allocation copies to another array of different allocation then transposition will be performed automatically in order to preserve indexes.

7.5 Primitive Communication

Primitive communication types provide for more optimised, granular communication control in Mesham than the default options provided. However, unlike the shared variable approach adopted elsewhere, when using primitive communication the programmer is responsible for ensuring communications complete and match up.

Channel

Syntax: channel[a,b]

Where *a* and *b* are both integer distinct processes which the channel will connect.

Semantics: The *channel* type will specify that a variable is a channel from process a (sender) to process b (receiver.) Normally this will result in synchronous communication, although if the *async* type is used then asynchronous communication is selected instead. Note that channel is unidirectional, where process a sends and b receives, NOT the otherway around.

Example:

```
var x:Int::allocated[multiple[]];
var p;
par p from 0 to 2
{
        (x::channel[0,2]):=193;
        var hello:=(x::channel[0,2]);
};
```

In this case, *x* is a channel between processes 0 and 2. In the par loop process 0 sends the value 193 to process 2. Then the variable *hello* is declared and process 2 will receive this value.

Notes: If no allocation information is specified with the channel type then the underlying variable is not assigned any memory. Instead, in this case, the variable can be thought of as a connector between two processes but has no physical existance. *Default:* Without further type information, all point to point explicit type communications are blocking using the standard send.

Pipe

pipe[a,b]

Identical to *channel*, except it is bidirectional rather than unidirectional

Onesided

onesided[a,b]

Very similar to channel, but will perform onesided communication rather than p2p. This form of communication is less efficient than p2p, but there are no issues such as deadlock to consider.

Reduce

Syntax: reduce[root,operation]

Semantics: All processes in the group will combine their values together at the root process and then the operation will be performed on them. Numerous operations are supported as detailed in table 7.

Example:

```
var t:Int::allocated[multiple[]];
var x:Int::allocated[multiple[]];
var p;
par p from 0 to 3
{
        x:(x::reduce[1,"max"]);
        x:=p;
        t:=x;
};
```

In this example, x is to be reduced, with the root as process 1 and the operation will be to find the maximum number. In the first assignment *x:=p* all processes will combine their values of p and the maximum will be placed into process 1's x. In the second assignment *t:=x* processes will combine their values of x and the maximum will be placed into process 1's t.

Operator	Description
max	Identify maximum value
min	Identify minimum value
sum	Compute sum of all values
prod	Compute product of all values

Table 7: Supported reduction operators

Broadcast

Syntax: broadcast[root]

Semantics: This type will broadcast a variable amongst the processes, with the root (source) being PID=root. The variable concerned must either be allocated to all or a group of processes (in the later case communication will be limited to that group.)

Example:

```
var a:Int::allocated[multiple[]];
var p;
par p from 0 to 3
{
        (a::broadcast[2]):=23;
};
```

In this example process 2 (the root) will broadcast the value 23 amongst the processes, each process receiving this value and placing it into their copy of a.

Gather

Syntax: gather[elements,root]

Semantics: Gather a number of elements (equal to *elements*) from each process and send these to the root process.

Example:

```
var x:array[Int,12]  ::  allocated[single[on[2]]];
var r:array[Int,3]  ::  allocated[multiple[]];
var p;
par p from 0 to 3
{
        (x::gather[3,2]):=r;
};
```

In this example, the variable x is allocated on the root process (2) only. Whereas r is allocated on all processes. In the assignment all three elements of r are gathered from each process and sent to the root process (2) and then placed into variable x in the order defined by the source's PID.

Scatter

Syntax: scatter[elements,root]

Semantics: Will send a number of elements (equal to *elements*) from the root process to all other processes.

Example:

```
var x:array[Int,3]::allocated[multiple[]];
var r:array[Int,12]::allocated[multiple[]];
var p;
par p from 0 to 3
{
        x:x::scatter[3,1];
        x:=r;
};
```

In this example, three elements of array *r*, on process 1, are scattered to each other process and placed in their copy of *r*.

Alltoall

Syntax: alltoall[elementsoneach]

Semantics: Will cause each process to send some elements (the number being equal to *elementsoneach*) to every other process in the group.

Example:

```
x:array[Int,12]::allocated[multiple[]];
var r:array[Int,3]::allocated[multiple[]];
var p;
par p from 0 to 3
{
        (x:alltoall[3]):=r;
};
```

In this example each process sends every other process three elements (the elements in its *r*.) Therefore each process ends up with twelve elements in *x*, the location of each is based on the source processes's PID.

Allreduce

Syntax: allreduce[operation]

Semantics: Similar to the *reduce* type, but the reduction will be performed on each process and the result is also available to all. Numerous operations are supported as detailed in table 7.

Example:

```
var x:Int::allocated[multiple[]];
var p;
par p from 0 to 3
{
        (x::allreduce["min"]):=p;
};
```

In this case all processes will perform the reduction on *p* and all processes will have the minimum value of *p* placed into their copy of *x*.

7.6 Communication Mode

By default, communication in Mesham is blocking (i.e. will not continue until a send or receive has completed.) Standard sends will complete either when the message has been sent to the target processor or when it has been copied into a buffer, on the source machine, ready for sending. In most situations the standard send is the most efficient, however in some specialist situations more performance can be gained by overriding this.

Asynchronous

Syntax: async[ ]

Semantics: This type will specify that the communication to be carried out should be done so asynchronously. Asynchronous communication is often very useful and, if used correctly, can increase the efficiency of some applications (although care must be taken.) There are a number of different ways that the results of asynchronous communication can be accepted, when the asynchronous operation is honoured then the data is placed into the variable, however when exactly the operation will be honoured is none deterministic and care must be taken if using dirty values.

The *sync* keyword allows the programmer to either synchronise ALL or a specific variable's asynchronous communication. The programmer must ensure that all asynchronous communications have been honoured before the process exits, otherwise bad things will happen!

Example:

```
var a:Int::allocated[multiple[]] :: channel[0,1] :: async[];
var p;
par p from 0 to 2
{
        a:=89;
        var q:=20;
        q:=a;
        sync q;
};
```

In this example, *a* is declared to be an integer, allocated to all processes, and to act as an asynchronous channel between processes 0 and 1. In the par loop, the assignment *a:=89* is applicable on process 0 only, resulting in an asynchronous send. Each process executes the assignment and declaration *var q:=20* but only process 1 will execute the last assignment *q:=a*, resulting in an asynchronous receive. Each process then synchronises all the communications relating to variable *q*.

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: async[];
var c:Int::allocated[single[on[3]]] :: async[];
a:=b;
c:=a;
b:=c;
sync;
```

This example demonstrates the use of the *async* type in terms of default shared variable style communication. In the assignment *a:=b*, processor 2 will issue an asynchronous send and processor 1 will issue a synchronous (standard) receive. The second assignment, *c:=a*, processor 3 will issue an asynchronous receive and processor 1 a synchronous send. In the last assignment, *b:=c*, both processors (3 and 2) will issue asynchronous communication calls (send and receive respectively.) The last line of the program will force each process to wait and complete all asynchronous communications.

blocking

Syntax: blocking[ ]

Semantics: Will force P2P communication to be blocking, which is the default setting

Example:

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: blocking[];
a:=b;
```

The P2P communication (send on process 2 and receive on process 1) resulting from assignment *a:=b* will force program flow to wait until it has completed. The *blocking* type has been omitted from the that of variable *a*, but is used by default.

nonblocking

Syntax: nonblocking[ ]

Semantics: This type will force P2P communication to be nonblocking. In this mode communication (send or receive) can be thought of as having two distinct states - start and finish. The nonblocking type will start communication and allows program execution to continue between these two states, whilst blocking (standard) mode requires the finish state has been reached before continuing. The *sync* keyword can be used to force the program to wait until finish state has been reached.

Example:

```
var a:Int::allocated[single[on[1]]] :: nonblocking[];
var b:Int::allocated[single[on[2]]];
a:=b;
sync a;
```

In the P2P communication resulting from assignment *a:=b*, process 1 will issue a non-blocking receive whilst process 2 will issue a blocking send. All nonblocking communication with respect to variable *a* is completed by the keyword *sync a*.

standard

Syntax: standard[ ]

Semantics: This type will force P2P sends to follow the standard form of reaching the finish state either when the message has been delivered or it has been copied into a buffer on the sender. This is the default applied if further type information is not present.

Example:

```
var a:Int::allocated[single[on[1]]] :: nonblocking[] :: standard[];
var b:Int::allocated[single[on[2]]] :: standard[];
a:=b;
```

In the P2P communication resulting from assignment *a:=b*, process 1 will issue a non-blocking standard receive whilst process 2 will issue a blocking standard send.

buffered

Syntax: buffered[buffersize]

Semantics: This type will ensure that P2P Send will reach the finish state (i.e. complete) when the message is copied into a buffer of size *buffersize* bytes. At some later point the message will be sent to the target process. If *buffersize* is not provided then a default is used.

Example:

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: buffered[500];
var c:Int::allocated[single[on[2]]] :: buffered[500] :: nonblocking[];
a:=b;
a:=c;
```

The P2P communication resulting from assignment *a:=b*, process 2 will issue a (blocking) buffered send (buffer size 500 bytes), which will complete once the message has been copied into this buffer. The assignment *a:=c*, process 1 will issue another send this time also buffered but nonblocking where program flow will continue between the start and finish state of communication. The finish state will be reached once the value of variable *c* has been copied into a buffer held on process 2.

ready

Syntax: ready[ ]

Semantics: The *ready* type will force P2P Send to start only if a matching receive has been posted by the target processor. When used in conjunction with the *nonblocking* type, communication start will wait until a matching receive is posted. This type acts as a form of handshaking and can improve performance in some uses.

Example:

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: ready[];
var c:Int::allocated[single[on[2]]] :: ready[] :: nonblocking[];
a:=b;
a:=c;
```

The send of assignment *a:=b* will only begin once the receive from process 1 has been issued. With the statement *a:=c* the send, even though it is nonblocking, will only start once a matching receive has been issued too.

synchronous

Syntax: synchronous[ ]

Semantics: By using this type, the send of P2P communication will only reach the finish state once the message has been received by the target processor.

Example:

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: synchronous[] :: blocking[];
var c:Int::allocated[single[on[2]]] :: synchronous[] :: nonblocking[];
a:=b;
a:=c;
```

The send of assignment *a:=b* (and program execution on process 2) will only complete once process 1 has received the value of *b*. The send involved with the second assignment is synchronous nonblocking where program execution can continue between the start and finish state, the finish state only reached once process 1 has received the message (value of *c*.) Incidentally, as already mentioned, the *blocking* type of variable *b* would have been chosen by default if omitted (as in previous examples.)

```
var a:Int :: allocated[single[on[0]]];
var b:Int :: allocated[single[on[1]]];
a:=b;
a:=(b :: synchronous[]);
```

The code example above demonstrates the programmer's ability to change the communication send mode just for a specific assignment. In the first assignment, process 1 issues a blocking standard send, however in the second assignment the communication mode type *synchronous* is coerced with the type of *b* to provide a blocking synchronous send just for this assignment only.

7.7 Partition

When the programmer partitions data, the compiler splits it up into blocks. The location of these blocks depends on the distribution type used - it is possible for all the blocks to be located on one process, on a few or on all and if there are more blocks than processes they can always "wrap around." The whole idea is that the programmer can refer to separate blocks without needing to worry about exactly where they are located, this means that it's very easy to change the distribution method to something more efficient later down the line if required.

The programmer can think of two types of partitioning - partitioning for distribution and partitioning for viewing. The partition type located inside the allocated type is the partition for distribution (and also the default view of the data.) However, if the programmer wishes to change the way they are viewing the blocks of data, then a different partition type can be coerced. This will modify the view of the data, but NOT the underlying way that the data is allocated and distributed amongst the processes. Of course, it is important to avoid an ambiguous combination of partition types. In order to access a certain block of a partition, simply use array access # or [ ] i.e. (a#3) will access the 3rd block of variable a.

In the code **var a:array[Int,10,20] :: allocated[A[m] :: single[D[]]];**, the variable *a* is declared to be a 2d array size 10 by 20, using partition type A and splitting the data into *m* blocks. These blocks are distributed amongst the processes via distribution method *D*.

In the code fragment **a:(a::B[])**, the partition type *B* is coerced with the type of variable *a*, and the view of the data changes from that of *A* to *B*.

Horizontal

Syntax: horizontal[blocks]

Where *blocks* is number of blocks to partition into.

Semantics: This type will split up data horizontally into a number of blocks. If the split is uneven then the extra data will be distributed amongst the blocks in the most efficient way in order to keep the blocks a similar size. The figure 2 illustrates the horizontal partitioning of an array into three blocks.

Assigned Variable	Assigning Variable	Semantics
single	partition	Gather
partition	single	Scatter
partition	partition	Local Copy

Table 8: Partition type communication in assignment

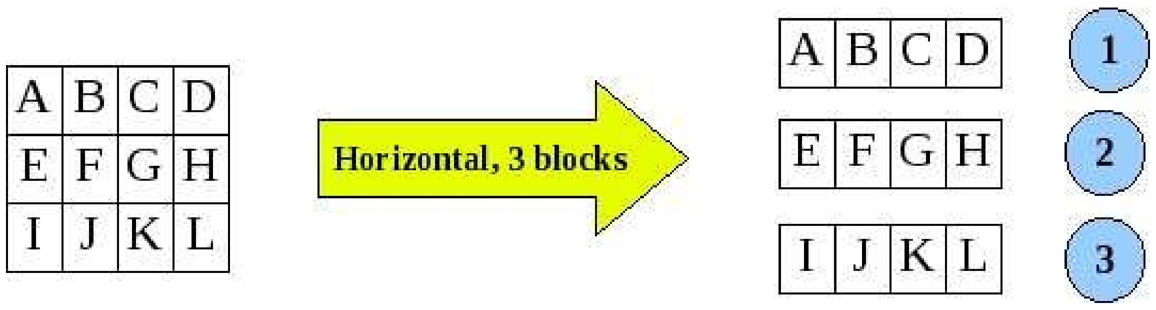

Figure 2: Horizontal Partitioning of data

Communication: There are a number of different default communication rules associated with the horizontal partition, based on the assignment *assigned variable:=assigning variable* which are detailed in table 8.

As in the last row of table 8, if the two partitions are the same type then a simple copy is performed. However, if they are different then an error will be generated as Mesham disallows differently typed partitions to be assigned to each other.

Horizontal blocks also support .high and .low, which will return the top and bottom bounds of the block

Vertical

Same as *horizontal*, but will partition vertically rather than horizontally. Figure 3 illustrates partitioning an array vertically into 4 blocks.

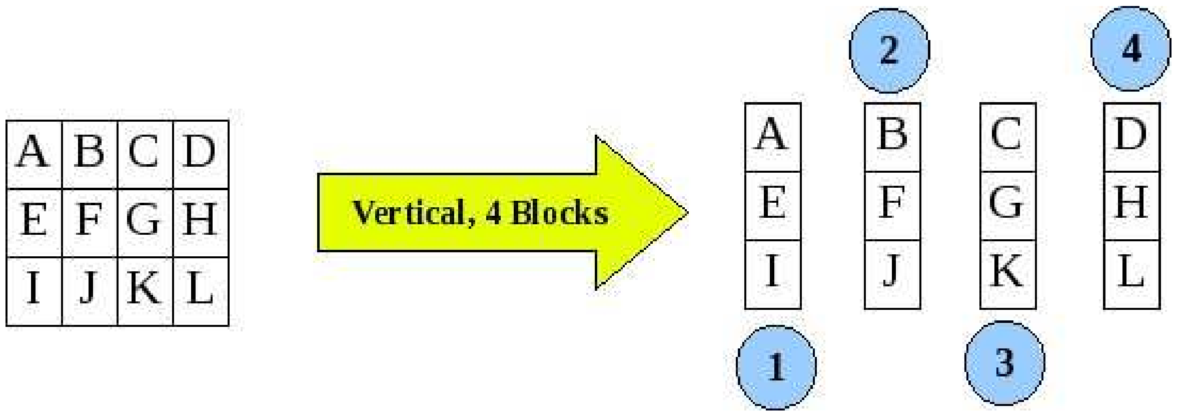

Figure 3: Vertical Partitioning of data

7.8 Distribution

Evendist

Syntax: evendist[ ]

Semantics: Will distribute data blocks evenly amongst the processes. If there are too few processes then the blocks will wrap around, if there are too few blocks then not all processes will receive a block. Figure 4 illustrates even distribution of 10 blocks of data over 4 processes.

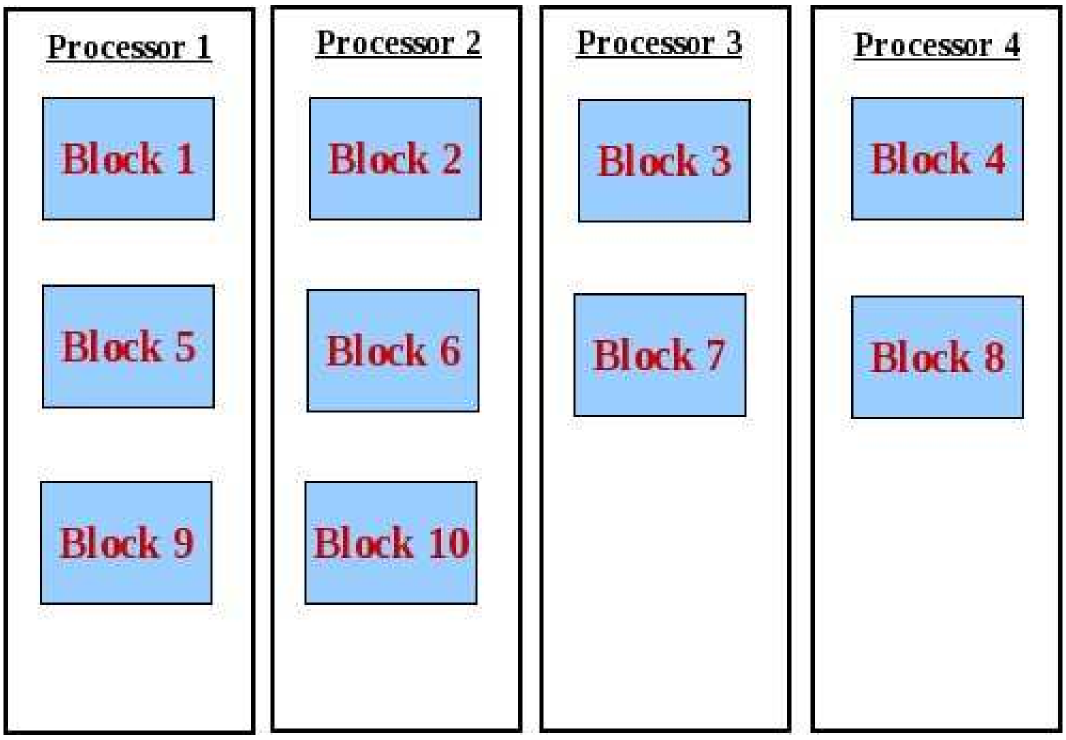

Figure 4: Even distribution of 10 blocks over 4 processes

7.9 Composition

Represents a structured, named, collection of element types. Compositions types differ from the collection type group by virtue of the fact that they contain named members. By default variables which are composition typed are allocated onto the heap.

Record

Syntax: record[name$_1$, type$_1$,name$_2$,type$_2$,.....,name$_d$,type$_d$]

Semantics: The *record* type allows the programmer to combine d attributes into one, new type. There can be any number of names and types inside the record type. A record type is very similar to a typedef structure in C. To access the member of a record use the dot, .

Example:

```
var complex : record["r",Float,"i",Float];
var person: record["name",String, "age",Int, "gender",Char];
var a:array[complex,10];
(a#1).i:=22.3;
var b:complex;
var me:person;
me.name:="nick";
```

In the above example, *complex* (a complex number) is a record with two *float* elements, *i* and *r*. The variable *b* is defined as a complex number and *a* as an array of these numbers. The variable *me* is of type *person*.

Notes: The *record* type should aim to maximise contiguous memory layout where ever possible.

Reference Record

Syntax: referencerecord[name_1, type_1,name_2,type_2,.....,name_d,type_d]

Semantics: The *record* type may NOT refer to itself (or other records) where as reference records support this, allowing the programmer to create data structures such as linked lists and trees. There are some added complexities of reference records, such as communicating them (all links and linking nodes will be communicated with the record) and freeing the data (garbage collection.) This results in a slight performance hit and is the reason why the record concept has been split into two types.

Example:

```
var node:referencerecord["prev",node,"Int",data,"next",node];
var head:node;
head:=null;
var i;
for i from 0 to 9
{
        var newnode:node;
        newnode.data:=i;
        newnode.next:=head;
        head.prev:=newnode;
        head:=newnode;
};

while (head != null)
{
        print(head.data,"\n");
        head:=head.next;
};
```

In this code example a doubly linked list is created, and then its contents read node by node.

Notes: The *referencerecord* type should aim to maximise contiguous memory layout where ever possible.

8 Function Library

By definition the language has available a function library which contains a minimal amount of critical functionality. This specified library is designed to be as simple and basic as possible. More complex functionality can be developed and supplied as part of third party libraries.

For access to a specific function then the programmer must include the appropriate function sub library within their code via the preprocessor directive. Table 9 details the in built sub libraries which should be included via the appropriate pre-processor directive.

Sub-library	Include file	Description
Maths	maths	Mathematical functionality
Input/Output	io	Input/Output support
Parallelism	parallel	Parallel specific functions
Bits	bits	Manipulation and processing at bit level
String	string	String handling, processing and manipulation
System	system	System level support and interaction

Table 9: Mesham function sub-libraries

8.1 Maths

cos

This cos(x) function will find the cosine of the value or variable x passed to it.

Pass: A double to find cosine of
Returns: A double representing the cosine

Example:

```
var a:=cos(10.0);
var y;
y:=cos(a);
```

sin

This sin(x) function will find the sine of the value or variable x passed to it.

Pass: A double to find the sine of
Returns: A double representing the sine

tan

This tan(x) function will find the tangent of the value or variable x passed to it.

Pass: A double to find the tangent of
Returns: A double representing the tangent

acos

This acos(x) function will find the inverse cosine of the value or variable x passed to it.

Pass: A double to find inverse cosine of
Returns: A double representing the inverse cosine

asin

This asin(x) function will find the inverse sine of the value or variable x passed to it.

Pass: A double to find inverse sine of
Returns: A double representing the inverse sine

atan

This atan(x) function will find the inverse tangent of the value or variable x passed to it.

Pass: A double to find inverse tangent of
Returns: A double representing the inverse tangent

cosh

This cosh(x) function will find the hyperbolic cosine of the value or variable x passed to it.

Pass: A double to find hyperbolic cosine of
Returns: A double representing the hyperbolic cosine

sinh

This sinh(x) function will find the hyperbolic sine of the value or variable x passed to it.

Pass: A double to find hyperbolic sine of
Returns: A double representing the hyperbolic sine

tanh

This tanh(x) function will find the hyperbolic tangent of the value or variable x passed to it.

Pass: A double to find hyperbolic tangent of
Returns: A double representing the hyperbolic tangent

floor

This floor(x) function will find the largest integer less than or equal to x.

Pass: A double to find floor of
Returns: An integer representing the floor

Example:

```
var a:=floor(10.5);
var y;
y:=floor(a);
```

ceil

The ceil(x) function will find the largest integer greater than or equal to x.

Pass: A double to find ceiling of
Returns: An integer representing the ceiling

getprime

This getprime(n) function will find the nth prime number.

Pass: An integer
Returns: An integer representing the prime

Example:

```
var a:=getprime(10);
var y;
y:=getprime(a);
```

log

This log(x) function will find the natural logarithmic value of *x*.

Pass: A double
Returns: A double representing the natural logarithmic value

Example:

```
var a:=log(10);
var y;
y:=log(a);
```

log10

This log10(x) function will find the base 10 logarithmic value of *x*.

Pass: A double
Returns: A double representing the base 10 logarithmic value

mod

This mod(n,x) function will divide n by x and return the remainder.

Pass: Two integers
Returns: An integer representing the remainder

Example:

```
var a:=mod(7,2);
var y;
y:=mod(a,a);
```

pi

This pi() function will return PI. *Note: The number of significant figures of PI is implementation specific.*

Pass: None
Returns: A double representing PI

Example:

```
var a:=pi();
```

pow

This pow(x,n) function will return *x* to the power of *n*.

Pass: A double (x) and an integer (n)
Returns: A double representing the result

Example:

```
var a:=pow(2,8);
```

randomnumber

This randomnumber(n,x) function will return a random number between n and x. *Note: A whole number will be returned UNLESS you pass the bounds of 0,1 and in this case a decimal number is found.*

Pass: Two integers defining the bounds of the random number
Returns: A float representing the random number

Example:

```
var a:=randomnumber(10,20);
var b:=randomnumber(0,1);
```

In this case, a is a whole number between 10 and 20, whereas b is a decimal number

sqrt

This sqrt(x) function will return the result of square rooting *x*.

Pass: A double to find square root of
Returns: A double which is the square root

Example:

```
var a:=sqrt(8.5);
```

8.2 Input/Output

close

This close(f) function will close the file represented by handle *f*.

Pass: A file handle of type File
Returns: Nothing

Example:

```
var f:=openfile("myfile.txt","r");
close(f);
```

input

This input(i) function will ask the user for input via stdin, the result being placed into *i*.

Pass: A variable for the input to be written into, of type String
Returns: Nothing

Example:

```
var f:String;
input(f);
print(f,"\n");
```

open

This open(n,a) function will open the file of name *n* with mode of *a*.

Pass: The name of the file to open type String and mode type String
Returns: A file handle of type File

Example:

```
var f:=open("myfile.txt","r");
close(f);
```

print

This print(n) function will write value or variable *n* to stdout.

Pass: String typed variable or string value to display
Returns: Nothing

Example:

```
var f:="hello";
print(f);
```

readchar

This readchar(f) function will read a character from a file with handle *f*. The file handle maintains its position in the file, so after a call to read char the position pointer will be incremented.

Pass: The file handle to read character from
Returns: A character from the file type Char

Example:

```
var a:=open("hello.txt","r");
var u:=readchar(a);
close(a);
```

readline

This readline(f) function will read a line from a file with handle *f*. The file handle maintains its position in the file, so after a call to readline the position pointer will be incremented.

Pass: The file handle to read the line from
Returns: A line of the file type String

Example:

```
var a:=open("hello.txt","r");
var u:=readline(a);
close(a);
```

write

This write(f,a) function will write the values of *a* to the file denoted by handle *f*.

Pass: The file handle to write to and also the value (any time) to write into file
Returns: Nothing

Example:

```
var a:=open("hello.txt","r");
write(a,"hello - test");
var q:=19;
write(a,q);
close(a);
```

8.3 Parallelism

pid

The pid() function will return the current processes' ID number.

Pass: Nothing
Returns: An integer representing the current process ID

Example:

```
var a:=pid();
```

processes

This processes() function will return the number of processes

Pass: Nothing
Returns: An integer representing the number of processes

Example:

```
var a:=processes();
```

8.4 Bits

bitreverse

This bitreverse(b,n) function will bit reverse the data held in *b* up to the number of elements *n*.

Pass: Data to bit reverse and an integer to of the number of elements held
Returns: Nothing

8.5 String

charat

This charat(s,n) function will return the character at position *n* of the string *s*.

Pass: A string and integer
Returns: A character

Example:

```
var a:="hello";
var c:=charat(a,2);
```

lowercase

This lowercase(s) function will return the lower case result of string *s*.

Pass: A string
Returns: A string

Example:

```
var a:="HeLlO";
var c:=lowercase(a);
```

strlen

This strlen(s) function will return the length of string *s*.

Pass: A string
Returns: An integer

Example:

```
var a:="hello";
var c:=strlen(a);
```

substring

This substring(s,n,x) function will return the string at the position between *n* and *x* of *s*.

Pass: A string and two integers
Returns: A string which is a subset of the string passed into it

Example:

```
var a:="hello";
var c:=substring(a,2,4);
```

toint

This toint(s) function will convert the string *s* into an integer.

Pass: A string
Returns: An integer

Example:

```
var a:="234";
var c:=toint(a);
```

tostring

This tostring(n) function will convert the variable or value *n* into a string.

Pass: Any element type (i.e. Integer, Float, Char, Double)
Returns: A string

Example:

```
var a:=234;
var c:=tostring(a);
```

uppercase

This uppercase(s) function will return the upper case result of string *s*.

Pass: A string
Returns: A string

Example:

```
var a:="HeLlO";
var c:=uppercase(a);
```

8.6 System

getepoch

This getepoch() function will return the number of milliseconds since the epoch (1st January 1970).

Pass: Nothing
Returns: Long containing the number of milliseconds since 1st January 1970

displaytime

This displaytime() function will display the timing results recorded by the function recordtime() along with the process ID. This is very useful for debugging or performance testing.

Pass: Nothing
Returns: Nothing

recordtime

This recordtime() function record the current execution time upon reaching that point. This is useful for debugging or performance testing, the time records can be displayed via displaytime().

Pass: Nothing
Returns: Nothing

exit

This exit() function will cease program execution and return to the operating system. From an implementation point of view, this will return EXIT_SUCCESS.

Pass: Nothing
Returns: Nothing

oscli

This oscli(a) function will pass the command line interface (e.g. Unix or MS DOS) command to the operating system for execution.

Pass: A string representing the command
Returns: Nothing

Example:

```
var a:String;
input(a);
oscli(a);
```

The above program is a simple interface, allowing the user to input a command and then passing this to the OS for execution.