1 Directives

All directives are present as text in a MATLAB comment and begin with keyword 'match' or 'MATCH'. All grammar specifications in this document follow YACC syntax. Lexical tokens are written in capitals. Since MATLAB comment is denoted by text starting with %, Each directive starts with a prefix string: %!match OR %!MATCH. Any unrecognized directives are ignored and a suitable warning message may be given to user. A directive may not continue on the next line. Multiple directives can be specified on one line by using ':;' as delimiter. The yacc grammar for the directives is as shown:

directive:
directive_prefix any_directive
;

directive_prefix:
  'match'
  | 'MATCH'
  
Here the string within ' is the text as seen by lexical analyzer.

any_directive :
type_directive
| data_distribution_directive
| data_alignment_directive
| task_directive
| data_precision
| processor_spec
| processor_type_spec
| task_begin_spec
| task_end_spec
;

As new directives are added, they would be added as alternates in above grammar. The individual directives are described in sections that follow. At present task directives have not been implemented.

2 Type and Shape Directive

Type and Shape directives unambiguously specify the type of MATLAB variable and its shape (scalar or array with number of elements in each dimension of the array).

2.1 Grammar

_type_directive:
  TYPE typeshape_name ident_list
  :
  type_name:
    REAL
    | INTEGER
    | UINTS
    | COMPLEX
    :

The typeshape_names are individual lexical tokens. A string 'real' or 'REAL' is returned as lexical token REAL above.

ident_list :
IDENTIFIER '( arglist ')' | ident_list ',' IDENTIFIER '( arglist )'
| IDENTIFIER
| ident_list ',' IDENTIFIER
;

arglist :
  value
  | arglist ',' value
  :
  value :
    INTEGER
    | IDENTIFIER
    | "unknown"
The lexical token IDENTIFIER refers to the name of a valid MATLAB scalar variable that takes integral value. The lexical token INTEGER refers to any valid positive, non-zero integer value represented as a string of decimal digits. Any number of spaces and tabs in comment text can be used to delimit individual lexical tokens. A unique string 'unknown' can be used to specify partial dimensionality (i.e. bounds in some (not all) dimensions are known) or unknown dimensionality. In the case of partial dimensions, number of dimensions are known.

2.2 Examples

% match TYPE integer a
a = 2 specifies 'a' as being a scalar
% match TYPE real a(512,512) , b(100)
[a, b] = foo(1, 2) specifies a as 512x512 real matrix, b is a 100 element row vector.
% match TYPE real a(512,512) , b(N)
[a, b] = foo(1, 2) Here N must be a constant known at compile-time.
% match TYPE real a(unknown), b(unknown,10)
[a, b] = foo(1, 2)
At present we do not support the use of an integer scalar variable to specify the number of elements in a specific dimension of an array variable, unless the value of such a variable (N in example above) is unambiguously known to be a specific positive integer value at compile time. However by using names of scalar integer variables, the bounds in shape of a specific dimension of an array can easily be parameterized. No arithmetic expression other than variable name or constant integral value are allowed in the specification. This restriction may be relaxed in later versions.

2.3 Semantics

- A variable name referred to in the type directive, and defined in the applicable MATLAB statement (see scope in next section) is asserted to have type and shape indicated in the directive. If no shape is indicated, the variable is assumed to be a scalar. An 'unknown' shape can be represented using '(unknown)'.
- All variable names in the type directive are case-sensitive.
- Use of a MATLAB variable name in specifying dimension does not assert that the variable is a scalar integer. This assertion must be separately made using TYPE directive for that variable. However it is required that the variable in question be scalar.
- Use of a MATLAB variable name having an ambiguous type and or shape (scalar, array etc) at the point before the MATLAB statement where the variable is used to specify a dimension is not allowed.

2.4 Scope

The MATLAB TYPE directive for a variable 'v' placed exactly before (in program's lexical order) a MATLAB statement that defines 'v', applies to that 'definition' of the variable v.

2.5 Additional Comments

- Definitions to more than one variable occur in MATLAB statements where a list of variables is assigned. e.g. [a,b,c] = foo(arguments), where a,b,c are return arguments of a function foo.
- The number of elements in each dimensions must be specified in the directive either as a constant integral value or as a MATLAB scalar integer variable.

3 Data Distribution

To be done

4 Data Alignment

To be done
5 Data Precision

Data precision directives assert that some properties are met by the data variables. If at runtime, the user assertion is false, results of program execution are undefined. In particular the number of bits required to specify a variable, in case of real variables, the size of mantissa, exponent etc. are specified. A data precision directive at a statement s refers to variable defined in the statement s as is the case with type and shape directives.

\[
data\_precision:
\text{bit\_spec attrib variable\_list ;}
\text{bit\_spec:BITS '\numeric'\prime ;}
\text{attrib: MANTISSA '\numeric'\prime opt\_exp opt\_radix;}
\text{opt\_exp: EXPONENT '\numeric opt\_bias'\prime ;}
\text{opt\_bias: NUMBER ;}
\text{opt\_radix: RADIX '\numeric'\prime ;}
\]

If no attrib is specified, the precision refers to integral value. Floating point numbers are described using mantissa, exponent, exponent bias and radix. The number in bit_spec refers to number of bits required to store a value. For floating point specification, exponent bias defaults to \(2^{\text{exponent}_{\text{unsigned}}-1} - 1\). If unspecified, radix is assumed to be binary.

Example:

\[
%!\text{match BITS}(32) \text{ a a=0; } \\
%!\text{Here a is a 32 bit variable.} \\
%!\text{match BITS}(32) \text{ MANTISSA}(24) \\
\text{EXPONENT}(8,127) \text{ RADIX}(2) \\
f=3.141592
\]

6 Destination Processor Type

A \text{processor\_spec\_type} directive allows a user to specify that a statement or task execute on a specific implementation defined architecture. Within a set of statements that are part of a task (see task directive), the processor type specification and processor number specification applies to the entire task. There cannot be two conflicting processor type (and number) specifications within a task. If a processor specification is lexically before a statement that does not belong to any task, it specifies a unique task consisting of a single statement. The chain of lexically adjacent program statements that do not belong to any task are considered to be a single task by the compiler.

\[
\text{processor\_spec:}
\text{USE number PROCESSORS} \\
\text{opt-processor-list ;}
\text{processor\_type\_spec:}
\text{USE PROCESSOR TYPE STRING ;}
\]

where the string refers to a valid architecture type. Currently we support "DSP", "FPGA", "PPC". This directive applies to the computation in the statement that lexically succeeds it. If the statement to which this directive applies, is part of a task, this directive applies to all the statements in the task. A statement may not have multiple processor_spec directives that applicable to it. If parts of an arithmetic expression are to be evaluated on different processor types, these expressions must be split into different statements by the programmer using temporaries.

Example:

\[
%!\text{match USE 4 PROCESSORS} \\
%!\text{match USE PROCESSOR TYPE 'dsp'} \\
a=\text{fft2}(\text{input});
\]

7 Specification of Tasks

\text{task\_begin\_spec:}
TASK begin IDENTIFIER
; task_end_spec:
    TASK end IDENTIFIER
;
A set of lexically consecutive statements constitutes a task. Tasks are demarcated by statements with task_begin_spec and task_end_spec directives. Tasks cannot be nested. All tasks need not execute. I.e. a task can be along a control flow dependent path that may not execute under some circumstances. To be precise, a task need not post-dominate the unique 'start' task (which is invoked on the beginning of the program execution). A task may not be specified within a user function. This restriction simplifies the analysis of tasks, and may be removed in later releases. A task may have multiple dynamic activations. I.e. the body of a loop can be specified as being a task. A single task can execute on more than one processors in SPMD fashion.

8 Parallelism & Serialization

A statement, or a group of statements can be marked as being serial (or parallel), by means of this directive. A statement or a set of statements enclosed within a parallel scope. At present these are used to mark a loop as having no dependences among iterations and each iteration can execute independent of each other. A group of data-parallel array oriented statements can be grouped in one parallel scope if there are no data dependences across these statements. A PARALLEL BEGIN starts a parallel scope and it must be terminated by a PARALLEL END. All statements within the parallel scope must be control post-dominated by the statement marking the end of parallel scope. In other words, if the statement with a parallel begin is encountered, it is statically decidable which statement terminates this parallel scope. If a statement $S_t$ that starts the parallel scope is executed, the statement that ends the parallel scope $S_e$ must always execute, and it must be encountered after the the statement $S_t$. The same restrictions apply for serial scope started by SERIAL BEGIN and terminated by SERIAL END.

parallelism_spec:
    SERIAL BEGIN
    | SERIAL END
    | PARALLEL BEGIN
    | PARALLEL END
    ;

9 Implementation Status

MATCH compiler updates the type and shape field of variable entries based on directives. Destination processor Type specification is currently supported by the compiler and is the primary mechanism of specifying execution of a statement on specified processor type. The data distribution and alignment directives are stored as 'post-it' data structure accessed by key pt.distribute.key. A pt.procmap.key key is used to specify virtual processor mesh. Statements are marked parallel by post-its pt.parallel.begin.key and pt.parallel.end.key placed on AST. List of serial statements are marked by keys pt.serial.begin.key and pt.serial.end.key. Data precision is specified by pt.fz.bits.key and pt.fp.bits.key tags. The number of processors in processor_spec directive are placed as pt.processor.count.key tag pt.arch.key tag holds processor type specification. Task beginning and end are marked by pt.task.begin.key and pt.task.end.key respectively.