

Appendix A The Compilable *Mathematica* Subset of MathCode F90

The *MathCode*¹ system provides facilities to translate a subset of the *Mathematica* language to compiled programs in strongly typed languages such as C++ or Fortran90, and in the future other languages like Java, etc. This subset includes most elementary functions and operators that compute numeric values, but excludes symbolic and computer algebra related functions that compute symbolic expressions.

However, it is possible to evaluate a symbolic expression (which may contain operations such as simplification, symbolic differentiation, substitution etc.) and generate executable numeric code from the symbolic expression resulting from this evaluation, provided that the resulting expression(s) only contain operators and functions in the compilable *Mathematica* subset described here.

The arithmetic model used in the compilable *Mathematica* subset is specified by the IEEE Standard for Binary Floating Point Arithmetic, IEEE Standard 754. Operations on complex numbers are currently not supported, but are planned in the near future.

A.1 Operations not in the Compilable Subset

The following is a short list of those *Mathematica* operations and functions that are not in the compilable subset. Since the primary reason to generate compiled code is to get high performance of numeric computing code, the operations in the compilable subset are oriented towards efficient computing on numbers and arrays.

- Pattern matching is not supported, except for the simple case of function argument patterns like `arg1_Integer` or `arg2_Real`, which are handled by the static type system of the target language. However, overloading of functions is not supported by the current version of the code generator, e.g. there may not be two functions with the

1. This chapter describes the *MathCode F90* release 1.0.1, May 2005

same name and arguments, one having `Integer` typed arguments and the other having `Real` typed arguments.

- When a function is declared, its arguments must be specified as single variable names, separated with commas. As an example, node patterns like `Name[a_, b_]` below are not permitted.

```
foo [Real[2] a_, Real c_]->Real[2] := ...      correct
fie [Real[2] Name[a_, b_], Real c_]->Real[2]:= ... incorrect
```

- Arbitrary precision numbers and arithmetic is not supported. Numbers and arithmetic operations are converted to either IEEE double precision floating point arithmetic or 32-bit (or better) integer arithmetic.
- Symbolic operations that give symbolic expressions as results are not included. However, such operations can be compiled if they are expanded to expressions in the compilable subset before code generation. Such expansion can handle many common cases of symbolic operations.
- Negative array indexing relative to the end of arrays are not in the compilable subset.
- String operations are not included.
- Input/Output operations are not included, apart from a simple `Print` operation.
- Certain list (i.e. array) operations, specifically certain operations that change the size of arrays or are very inefficient, are not included in the set of functions mentioned in this appendix. Such functions can be added by the user e.g. in the `system` module.
- The `Return[]` function is not included. Therefore loop constructs like `For`, `While` cannot be used as expressions returning values.
- Some procedural style statements cannot be used within a `CompoundExpression` used in value context within arithmetic expressions. For instance, `a=a+(While[i<10,i=i+1];5)` cannot be translated. The expression `a=a+(c=3;5)`, however, can be translated to C++. More details on nested constructs are given below.

A.2 Predefined Functions and Operators

Expression operators listed in this section are predefined by the code generator and will be translated correctly from *Mathematica* into the target language (e.g. C++ or Fortran90) without any additional type declarations.

Almost all operators belong to the *compilable expression subset*, i.e. all value-returning operators and predefined or user-defined functions without *side effects* (i.e. functions that do

not change global variables or perform input/output).

The reason to impose the condition of calls to side-effect free functions is that expressions can be re-ordered and common subexpressions removed in the generated code, in order to make execution more efficient. Another order in assigning and referencing global variables or performing input/output usually results in different, often unintended, program behavior. However, some restricted cases of side-effects can be re-ordered without changing the meaning of the program. One such case is when the elements of an array are assigned once, and independently of each other, and not used in the same expression. Such restricted side-effects are allowed for functions in the compilable expression subset. The code generator does not check the condition of side-effect freeness—this is the user’s responsibility.

All operators and functions in the compilable expression subset also belong to the compilable subset, which also contains control expressions (`If`, `While`, `For`, etc.), assignment statements and functions with side effects. All real and integer constants naturally belong to the compilable expression subset, except for the special case of arbitrary-precision values. Some operators and functions can be applied to arrays or return arrays as values.

The current version of the compilable subset is oriented toward operations on real numbers and integers, and arrays containing such numbers. The basic mathematical functions usually found in C/C++ or Fortran are provided. In *Mathematica* there are also a number of special mathematical functions such as `BesselJ[]`, `Gamma[]`, etc. If the user has access to an implementation of such a function in C/C++ or Fortran, or a linkable object code library containing this function, it can be declared as an external function and thus automatically included in the compilable subset.

Since efficient computation based on mathematical models so far has been the main application of MathCode, the compilable *Mathematica* subset does not include string operations, file input, formatted file output and certain mapping and list operations.

A.2.1 Statements and Value Expressions

In standard *Mathematica* all predefined and user-defined functions can appear as an argument of another function. Correctness of such constructs is tested during code interpretation.

In procedural languages, such as C++ and Fortran, procedural statements cannot be used within expressions. Also, the type of allowed expressions is restricted.

In order to compile *Mathematica* code to procedural language some restrictions in using statements and expressions are introduced.

In the descriptions below “*stmt*” means that corresponding *Mathematica* expressions are used as statements. In the compiled subset they do not return values, their returned values cannot be used, and they cannot be applied where values are expected. In the compiled set there is no `Null` value.

In descriptions below “*expr*” means that corresponding *Mathematica* expressions are used as values (l-value or r-value). These expressions must return some value when evaluated. This value cannot be Null. The word “*exprs*” means one or more expressions separated by a comma.

Some *Mathematica* constructs - Set, If, Which, CompoundExpression - can appear both as statements and as values. Some specific restrictions on their use are described below.

A.2.2 Function Call

Spec syntax	Operator	Arg type(s)	Result type(s)
	funcname[<i>exprs</i>]

All user-defined functions which have been *type* declared according to the typing rules for typed *Mathematica* belong to the compilable subset. Compilable subset functions may only contain operations that belong to the compilable subset, or may contain non-subset operations inside bodies of functions compiled with the EvaluateFunction option, which will expand into compilable subset operations.

Functions with *multiple* return arguments can be compiled if they are type declared. Such a function can only be used in the right hand side of an assignment statement in which the left hand side has to be a list of variables. Thus, a call that returns multiple values can for example look like this:

```
{a, b, c} = F[x+y, 3.4];
```

Calls to functions with *no return arguments* and functions with *more than one* return arguments are considered as statements (*stmt*).

Calls to functions returning one argument are considered as expressions (*expr*).

A.2.3 Function Definition

A function returning values can be defined as follows:

```
function_name[arg_type1 arg1, ..., arg_typen argn]->result_types :=
  expr
```

```
function_name [arg_type1 arg1, ..., arg_typen argn]->
  result_types := Module[variables, expr]
```

```
function_name [arg_type1 arg1, ..., arg_typen argn]->
  result_types := Module[variables, stmt1;stmt2;...;expr]
```

A function that does not return values can be defined as follows:

```
function_name[arg_type1 arg1, ..., arg_typen argn]->Null := stmt
```

```
function_name[arg_type1 arg1, ..., arg_typen argn]->Null :=
  Module[variables, stmt;]
```

```
function_name[arg_type1 arg1, ..., arg_typen argn]->Null :=
  Module[variables, stmt1; stmt2;...;stmtn;]
```

Block or With can be used instead of Module.

A.2.4 Scope Constructs

Spec syntax	Operator	Arg type(s)	Result type(s)
	Module[<i>variables,body</i>]	special	none/(fnbody)
	Block[<i>variables, body</i>]	special	none/(fnbody)
	With[<i>variables,body</i>]	special	none/(fnbody)

A value can be returned from one of the above scope constructs when it occurs as a function body or when it is used in value context within an expression. The *body* is restricted as follows:

- If a function does not return any value, the *body* is a statement. If it is a CompoundExpression statement, then all (possibly nested) elements in CompoundExpression must be statements.

In the following example two nesting levels of CompoundExpression are demonstrated:

```
foo[Real a_]->Null := Module[{ Real t},
  (t=a+1;t=t+1);(t=t+2;t=t+3)]
```

- If a function returns one or more values, the *body* is an expression. If it is a CompoundExpression construct, then the last (possibly nested) element in CompoundExpression must be an expression. All other components must be

statements.

In the following example two nesting levels of `CompoundExpression` are demonstrated; note that `t+4` is an expression.

```
foo[Real a_]->Real := Module[{ Real t },
  (t=a+1;t=t+1);(t=t+2;t=t+3;t+4) ]
```

A.2.5 Control Statements

The control statements can appear wherever a statement is allowed, in which case they do not return any value.

Spec syntax	Operator	Arg type(s)	Result type(s)
$s_1; s_2; \dots$	<code>CompoundExpression [stmts]</code>	statements	none
	For [<i>start-stmt</i> , <i>boolean-test-expr</i> , <i>incr-stmt</i> , <i>body-stmt</i>]	special	none
	While [<i>boolean-test-expr</i> , <i>body-stmt</i>]	special	none
	If [<i>boolean-test-expr</i> , <i>true-stmt</i> , <i>false-stmt</i>]	special	none
	Which [<i>boolean-test-expr</i> ₁ , <i>stmt</i> ₁ <i>boolean-test-expr</i> ₂ , <i>stmt</i> ₂ ,...]]	special	none
	Break []	-	none
	Do [<i>expr</i> , iterators]	special	none

`CompoundExpression` (a sequence of expressions separated by semicolon), `Which` and `If` can also appear as arithmetic expression. See “Arithmetic expression” for details.

A.2.6 Mapping Operations

`Map` expressions can be compiled in the following cases:

```
var=Map[ f , expr ]
var=Map[ f , expr , {n} ]
```

The result must be directly assigned to a variable as shown. The function `f` can be:

- A function symbol of the compilable subset
- An anonymous function, also called pure function in *Mathematica*
- A user defined typed function for which code has been generated

`n` must be an integer constant. The `var=Map[...]` statement will be converted to a corresponding assignment statement with a call to `Table` on the right-hand side.

A.2.7 Iterator Expressions

Computing operations in *Mathematica* such as `Do`, `Sum`, `Product` and `Table` use iterators. Additionally there are a number of plotting functions such as `Plot`, `ContourPlot`, `DensityPlot`, `Plot3D`, `ParametricPlot`, also using iterators but with some limitations in form and usually constructing sets of real values for the purpose of plotting. These plotting functions are not part of the compilable subset.

An iterator can take one of the following forms:

Form	Explanation
<code>{imax}</code>	iterate <i>imax</i> times
<code>{i,imax}</code>	<i>i</i> goes from 1 to <i>imax</i> in steps of 1
<code>{i,imin,imax}</code>	<i>i</i> goes from <i>imin</i> to <i>imax</i> in steps of 1
<code>{i,imin,imax,di}</code>	<i>i</i> goes from <i>imin</i> to <i>imax</i> in steps of <i>di</i>
<code>{i,imin,imax},{j,jmin,jmax}</code>	Two iterators: <i>i</i> controls the outer iteration loop, <i>j</i> controls the inner loop

Iterators in *Mathematica* can use either integer or real values for the iteration variables in the iteration. The compilable subset of iteration functions is limited to integer iteration variables. The iteration variables in *Mathematica* are declared in a local scope consisting of the body (the *expr* below) of the iteration function. Thus, translated code in Fortran90 needs to declare those iteration variables in a way that does not clash with other local variables. Typically, these iteration constructs will be translated to (nested) `for` loops in the target language.

Iteration functions in *Mathematica* may or may not return a value. The functions `Sum`, `Product`, `Table` and `Range` always return a value from the iteration. Loop-terminating constructs like `Return`, `Break`, `Continue`, or `Throw` can be used inside `Do`. However, `Do` in *Mathematica* does not return a value except in the case of an explicit `Return` of a value.

The compilable subset currently does not support return of a value from a `Do` loop. Another constraint of the compilable subset is that the constructs `Sum`, `Product` and `Table` may currently only occur on the right hand side of an assignment statement. Concerning `Table`, see also Section A.2.12.

Spec syntax	Operator	Arg type(s)	Result type(s)
	<code>Do[expr,iter1,iter2,...]</code>	special	none
	<code>Sum[expr,iter1,iter2,...]</code>	Real, Integer	Real, Integer
	<code>Product[expr,iter1,iter2,...]</code>	Real, Integer	Real, Integer
	<code>Table[expr,iter1,iter2,...]</code>	Real,Integer	Array

A.2.8 Input/Output Operations

Spec syntax	Operator	Arg type(s)	Result type(s)
	Print[<i>exprs</i>]	Real, Integer, Array, String, none	

The output is placed on the standard output stream of the external process where the generated code is executing. The recommended way to perform formatted input/output from generated code is via callback functions or external functions.

A.2.9 Standard Arithmetic and Logic Expressions

Spec syntax	Operator	Arg type(s)	Result type(s)
===	SameQ[e_1, e_2]	Real, Integer, Array	Boolean
===	UnSameQ[e_1, e_2]	Real, Integer, Array	Boolean
==	Equal[e_1, e_2]	Real, Integer, Array	Boolean
!=	Unequal[e_1, e_2]	Real, Integer, Array	Boolean
>	Greater[e_1, e_2]	Real, Integer	Boolean
<	Less[e_1, e_2]	Real, Integer	Boolean
>=	GreaterEqual[e_1, e_2]	Real, Integer	Boolean
<=	LessEqual[e_1, e_2]	Real, Integer	Boolean
	Inequality[<i>exprs...</i>]	<i>special</i>	Boolean
!	Not[e]	Boolean	Boolean
	Or[<i>exprs...</i>]	Boolean	Boolean
&&	And[<i>exprs...</i>]	Boolean	Boolean
+	Plus[<i>exprs...</i>]	Real, Integer, Array	Real, Integer, Array
-	Subtract[e_1, e_2]	Real, Integer, Array	Real, Integer, Array
-	Minus[e]	Real, Integer, Array	Real, Integer, Array
*	Times[<i>exprs...</i>]	Real, Integer, Array	Real, Integer, Array
/	Divide[e_1, e_2]	Real, Integer, Array	Real, Array
	Mod[e_1, e_2]	Real, Integer, Array	Real, Array
	Rational[e_1, e_2]	Real, Integer	Real
^	Power[e_1, e_2]	Real, Integer, Array	Real, Integer, Array
	Abs[e]	Real, Integer, Array	Real, Integer, Array
	If[<i>boolean-test-expr</i> , <i>true-expr>false-expr</i>] ¹	1st arg Boolean; Real, Integer, Array	Real, Integer, Array
	Sign[e]	Real, Integer, Array	Integer, Array
	Floor[e]	Real, Array	Integer, Array
	Ceiling[e]	Real, Array	Integer, Array
	Round[e]	Real, Array	Integer, Array

1. Read Release Notes for more information

	Sqrt[<i>e</i>]	Real, Integer, Array	Real, Array
	Exp[<i>e</i>]	Real, Integer, Array	Real, Array
	Log[<i>e</i>]	Real, Integer, Array	Real, Array
	Sin[<i>e</i>]	Real, Integer, Array	Real, Array
	Cos[<i>e</i>]	Real, Integer, Array	Real, Array
	Tan[<i>e</i>]	Real, Integer, Array	Real, Array
	Cot[<i>e</i>]	Real, Integer, Array	Real, Array
	Sec[<i>e</i>]	Real, Integer, Array	Real, Array
	Csc[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcSin[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcCos[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcTan[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcTan[<i>e</i> ₁ , <i>e</i> ₂]	Real, Integer, Array	Real, Array
	Sinh[<i>e</i>]	Real, Integer, Array	Real, Array
	Cosh[<i>e</i>]	Real, Integer, Array	Real, Array
	Coth[<i>e</i>]	Real, Integer, Array	Real, Array
	Sech[<i>e</i>]	Real, Integer, Array	Real, Array
	Csch[<i>e</i>]	Real, Integer, Array	Real, Array
	Tanh[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcSinh[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcCosh[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcTanh[<i>e</i>]	Real, Integer, Array	Real, Array
	ArcCoth[<i>e</i>]	Real, Integer, Array	Real, Array
	IntegerPart[<i>e</i>]	Real, Integer, Array	Integer, Array
	FractionalPart[<i>e</i>]	Real, Integer, Array	Real, Integer, Array
	Quotient[<i>e</i> ₁ , <i>e</i> ₂]	Real, Integer, Array	Integer, Array
	Max[<i>m</i> , <i>n</i>]	Real,Integer	Real,Integer
	Min[<i>m</i> , <i>n</i>]	Real,Integer	Real,Integer
	Max[<i>e</i>]	Array of Real	Real
	Max[<i>e</i>]	Array of Integer	Integer
	Min[<i>e</i>]	Array of Real	Real
	Min[<i>e</i>]	Array of Integer	Integer
	Outer[<i>e</i> ₁ , <i>e</i> ₂]	1D-Array,1D-Array	Array
	Cross[<i>e</i> ₁ , <i>e</i> ₂ ,..., <i>e</i> _{<i>n</i>}]	Arrays	Array
	Transpose[<i>e</i>]	2D-Array	2D-Array
<i>e</i> ₁ · <i>e</i> ₂ ...	Dot[<i>e</i> ₁ , <i>e</i> ₂ ,...]	Array	Real, Integer, Array
	CompoundExpression[<i>stmt</i> ₁ , ..., <i>stmt</i> _{<i>n</i>} , <i>expr</i>] ¹	statements	expr

1. Read Release Notes for more information

For functions with two arguments the following rule applies: one argument can be Array and another argument can be either scalar (of the same type as the base type of the array) or Array of the same dimension. This does not apply to == and !=.

The functions which return an integer value converted from real: Sign, Floor, Ceiling, Round, give an undefined value or an exception (depending on the underlying target language, e.g. Fortran90) when trying to fit too big a number into an integer.

The following functions are implemented according to *Mathematica* semantics¹:

- IntegerPart returns `(int)x`
- FractionalPart returns `x-int(x)`
- Quotient[m,n] returns `Floor(m/n)`

`Mod[m,n]` returns `m%n` if `m` and `n` have the same sign and `m%n+n` if they have opposite sign. If `m` or `n` is a Real then `m-n*floor(m/n)` is returned.

The Rational function is part of the compilable subset. It is treated exactly like Divide, and converted to Divide during code generation.

The special purpose Cross function computes the cross product of $n-1$ vectors of length n and returns vector of length n . For example, `Cross[{2,3,4},{5,6,7}]` returns the vector `{-3,6,-3}` which is orthogonal to the two argument vectors. The function Cross is implemented for $n=3,4,5$ according to the generalized *Mathematica* definition.

The CompoundExpression construct when used as a value within another statement or expression (but not as a function definition) has the following limitation: the statements ($stmt_1, \dots, stmt_n$) allowed within CompoundExpression are assignments (Set), Print or Put only.² Assignment to list cannot be used there. For instance:

```
a=b+(While[i<10,i=i+1] ; c); (* not allowed *)
a=Foo[{d,f}={3,5} ; c ]; (* not allowed *)
a=b+(Print[x];c); (* allowed *)
foo[Real a_]->Real=(i=i-1;(While[i<10,i=i+1] ; c)) // allowed
```

A.2.10 Named Constants

Spec syntax	Operator	Arg type(s)	Result type(s)
	True	-	Boolean

1. Read Release Notes for more information
 2. Read Release Notes for more information

False	-	Boolean
E	-	Real
Pi	-	Real

Variables of type `Boolean` are not supported in the compilable subset. If boolean values are assigned to integer variables, `False` becomes 0, `True` becomes non-zero. Named constants are expressions (*expr*).

A.2.11 Assignment Expressions

Spec syntax	Operator	Arg type(s)	Result type(s)
<code>var := e</code>	<code>SetDelayed[<i>var</i>,<i>e</i>]</code>	all types	value
<code>var = <i>expr</i></code>	<code>Set[<i>var</i>,<i>expr</i>]</code>	all types	value
<code>{<i>vars</i>} = <i>funcall</i></code>	<code>Set[List[<i>vars</i>],<i>funcall</i>]</code>	-	none
<code>{<i>vars</i>} = <i>expr</i></code>	<code>Set[List[<i>vars</i>],<i>expr</i>]</code>	-	none

The supported main assignment functions, `Set` and `SetDelayed`, have return types. Therefore these can be used both as statements and as expressions.

The arguments (left- and right-hand side of the assignment) must be of compatible types.

Left and right hand side arguments are compatible if they can be made the same type by performing standard type promotion (e.g. promoting integer to real, or a scalar or lower-dimensional array to a higher-dimensional array), provided that this promotion does not change the type of the left-hand side. If it does, then the assignment is illegal. This means that an expression of a real type cannot be assigned to a variable of integer type without using explicit conversion of the right-hand side (e.g. using `Floor[]`).

In the case of simultaneous assignment to a list of variables `{vars}`, *funcall* must be a call to a function returning a list of the same length as the list in the left hand side of the assignment. Also, the *vars* list in the left hand side must only contain variables.

A.2.12 Array Data Constructors

Spec syntax	Operator	Arg type(s)	Result type(s)
	<code>Array[<i>exprfunc</i>,{<i>dim1</i>,<i>dim2</i>,...}]</code>	<i>exprfunc</i> constant	Array
	<code>Table[<i>expr</i>,{<i>dim1</i>},{<i>dim2</i>},...]</code>		Array
	<code>Table[<i>expr</i>,{<i>i</i>,<i>imin</i>,<i>imax</i>,<i>istep</i>},{<i>j</i>,<i>jmin</i>,<i>jmax</i>,<i>jstep</i>},...]</code>		Array
	<code>IdentityMatrix[<i>n</i>]</code>	Integer	Array (2D)
	<code>DiagonalMatrix[<i>vec</i>]</code>	Array (1D)	Array (2D)
	<code>Range[<i>n</i>]</code>	Real or Integer	Array (1D)
	<code>Range[<i>start</i>,<i>end</i>]</code>	Real or Integer	Array (1D)
	<code>Range[<i>start</i>,<i>end</i>,<i>step</i>]</code>	Real or Integer	Array (1D)

See also section A.2.7 concerning iterator expressions. The following limitations currently

apply to compilation of `Array`, `Table`, `IdentityMatrix` and `DiagonalMatrix` calls: the *exprfunc* used by `Array` may only be a constant function; local iteration variables used in iterators to `Table` are automatically created but are always of type `Integer`; calls to `Array`, `Table`, `IdentityMatrix` and `DiagonalMatrix` may only occur at the right hand side of an assignment statement, for example:

```
arrvariable = Table[3.1+i+j, {i,5}, {j,1,10,2}]
```

A.2.13 Array Dimension Functions

Spec syntax	Operator	Arg type(s)	Result type(s)
	<code>Dimensions[arr][[i]]</code>	Array	Integer
	<code>Dimensions[arr]</code>	Array	Array of Integers
	<code>Length[arr]</code>	Array	Integer

A.2.14 Array Indexing

Spec syntax	Operator	Arg type(s)	Result type(s)
<code>arr[[ind]]</code>	<code>Part[arr,ind]</code>	Integer	Integer,Real,Array
	<code>Extract[a1,a2]</code>	Array; Integer	Element-type

`Extract[a, i]` takes an array of rank 1,2,3, or 4 as first argument and a vector of integers as the second argument. It returns the base element of the first array. If the size of the vector `i` is not equal to the rank of `a` then a run time error may occur.

The `Part` construct can be used in the left part and in the right part of assignment. The number of indices should be less or equal to the rank of the array. For instance, these operations are allowed:

```
Declare[
  Real[3,3,3,3] a4;
  Real[3,3,3] a3;
  Real[3,3] a2;
  Real[3] a1;
  Real x;
  ...
]

a3[1]=a2; a3[2,1]=a1; a3[3,1,2]=5.5;
a2[1]=a1; a2[2,2]=7.7;
a4[2,3,1,2] = 6.6;
```

```
x=Extract[a4,{2,3,1,2}]

a4[1,2,3]=a1;a4[1,2]=a2;a4[1]=a3;
```

This operation is not permitted:

```
a1=Extract[a4,{2,3,1}] (* Wrong rank. May cause run time error *)
```

A.2.15 Array Section Operations

Spec syntax	Operator	Arg type(s)	Result type(s)
arr[_]	Part[arr,...]	special	Array
arr[[n ₁ _]]	Part[arr,...]	special	Array
arr[[n ₁ n ₂]]	Part[arr,...]	special	Array

These are extensions to standard *Mathematica*. See Chapter 3 for more information. These operations are currently supported for up to four dimensions by the code generator and for arbitrary dimensions within *Mathematica*, and can be used on both the left-hand-side and right-hand-side of assignment statements.

A.2.16 Other Expressions

Spec syntax	Operator	Arg type(s)	Result type(s)
{e ₁ , e ₂ ,...}	List[expressions]	all types	Array
	Apply[f, args]		

List

List is partially implemented when appearing within expressions, for instance when used as an actual parameter to a function. The arguments of List can be:

- Real expressions (creates Array of Real)
- Integer expressions (creates Array of Integer)
- Arrays of Real (creates 2-, 3-, 4-dimensional Array of Real). Can be nested.
- Arrays of Integers (creates 2-, 3-, 4-dimensional Array of Integers). Can be nested.

List is also implemented when it appears on the left-hand-side of assignments. In this case Part is applied to the right-hand side, and all types should match¹:

1. Read Release Notes for more information

```
{a,b,{c,d}}=x (* is the same as
                a=x[[1]];b=x[[2]];c=x[[3,1]];d=x[[3,2]]; *)
```

Runtime error may occur if matrix appears to be non-rectangular.

These special cases are implemented:

```
variable={expr1,...,exprn}
```

```
{var1,...,varn}=expression
```

```
{var1,...,varn}={expr1,...,exprn}
```

Apply

The following cases of `Apply` are implemented:

- Plus, Power and Times applied to an expression with assignment to a typed variable:

```
var=Apply[Plus,expression]      var = Plus @@ expression
var=Apply[Power,expression]    var = Power @@ expression
var=Apply[Times,expression]    var = Times @@ expression
```

- Apply of typed functions, for example

```
var=Apply[function,expression]  function @@ expression
```

The number of arguments to the function must match the length of the expression.

- Apply of anonymous (pure) functions, for example

```
var=Apply[Sin[#1+#2]&,expression] Sin[#1+#2]& @@ expression
```

The code `Apply[foo,expr]`, equivalent to `foo @@ expr`, will be converted to `foo[expr[[1]],expr[[2]],...]`. Therefore the behaviour will be different from that of *Mathematica* (and hence probably unexpected) if the number of parameters is not the same as the length of the expression `expr`.

It is the user's responsibility to ensure that the number of arguments to the pure function is the same as the length of the expression. The number of arguments is taken as the maximum slot number (for `Function[body]`) or the length of the variable list (for `Function[{vars...}, body]`).

The expression given to `Apply` may be computed many times which may be a performance issue. If the expression is big it is better to assign the expression to a temporary variable before using `Apply`.

No level specification is supported for `Apply`.

A.2.17 Operators Which May Have Side-effects

Spec syntax	Operator	Arg type(s)	Result type(s)
$var := e$	SetDelayed[<i>var</i> , <i>e</i>]	all types	none
$var = expr$	Set[<i>var</i> , <i>expr</i>]	all types	none
$\{vars\}=funcall$	Set[List[<i>vars</i>], <i>funcall</i>]	special	none
	For[<i>start</i> , <i>test</i> , <i>incr</i> , <i>body</i>]	special	none
	While[<i>test</i> , <i>body</i>]	special	none
	Do[<i>expr</i> ,{ <i>iter1</i> ...},{ <i>iter2</i> ...}..]	special	none
	If[<i>test</i> , <i>true-expr</i> , <i>false-expr</i>]	special	none/expr
	If[<i>test</i> , <i>true-expr</i>]	special	none/expr
	Which[<i>test</i> ₁ , <i>val</i> ₁ , <i>test</i> ₂ , <i>val</i> ₂ ,...]	special	none
$e_1; e_2; ..$	Break[]	-	none
	CompoundExpression[<i>exprs</i>]	special	Real, Integer, Boolean
	Module[<i>variables</i> , <i>body</i>]	special	none/function value
	Block[<i>variables</i> , <i>body</i>]	special	none/function value
	With[<i>variables</i> , <i>body</i>]	special	none/function value

A.3 Predefined Types

As already mentioned, there are a number of predefined basic types included in the compilable subset of *Mathematica*. There is also a set of predefined types, primarily array types, which are included for convenience.

A.3.1 Basic Types

Name	Comment
Real	IEEE double precision floating point
Integer	32 bit integer
String	8-bit byte string. May contain '\0' characters.
Null	Absence of type

A.3.2 Array Type Constructors

Name	Comment
$eltype[\text{dim1},\text{dim2},\dots]$	Here <i>eltype</i> is the array type constructor.

Maximal rank of arrays is 4 in the current implementation. The base type should be Real or Integer.

A.4 Predefined Constants

The following constants are available within *Mathematica*, and are predefined to the following values with 18 decimal digits within generated C++ or Fortran90 code. A standard double precision floating point value can hold slightly less than 16 digits of precision.