
MetaSL

Shading Language Tutorial and Reference

Andy Kopra and Lutz Kettner

Excerpt:

Chapter 2: Elements of the MetaSL language

Preliminary draft — subject to change.

For latest version, see <http://www.metasl.org>

Copyright Information

Copyright © 1986-2011 mental images GmbH, Berlin, Germany.

All rights reserved.

This document is protected under copyright law. The contents of this document may not be translated, copied or duplicated in any form, in whole or in part, without the express written permission of mental images GmbH.

The information contained in this document is subject to change without notice. mental images GmbH and its employees shall not be responsible for incidental or consequential damages resulting from the use of this material or liable for technical or editorial omissions made herein.

mental images®, mental ray®, mental matter®, mental mill®, mental queue®, mental cloudTM, mental mesh®, RealityServer®, RealityPlayer®, RealityDesigner®, MetaSL®, Metanode®, Phenomenon®, neuray®, iray®, DiCETM, imatter®, Shape-By-Shading®, SPM®, and rendering imagination visibleTM are trademarks or, in some countries, registered trademarks of mental images GmbH, Berlin, Germany.

Other product names mentioned in this document may be trademarks or registered trademarks of their respective companies and are hereby acknowledged.

Chapter 2

Elements of the MetaSL language

The previous chapter described the large-scale structures of MetaSL that use the shader as a fundamental building block. This chapter proceeds in the opposite direction, starting with the smallest component of a MetaSL program, working up through the hierarchy of language features to present the internal structure of a shader.

2.1 Variables

MetaSL uses various data types derived from C++, but also defines types like `Color3` and `Ray` specific to the domain of shader programming. Variable declaration and initialization for scalar and array types are based on the syntax of C++. A constructor notation for the initialization of aggregate types like `Color3` is based on C++ class constructor syntax. Like C, aliases for type names can be created using the `typedef` specifier.

2.1.1 Values and data types

A *value* is a piece of data used by MetaSL in its calculations, like a number or a color. A value is characterized by its *data type*, a category to which the value belongs. These MetaSL data types are more precisely defined than the intuitive concepts on which the data types are based. For example, there is a difference in MetaSL between a number that you would use for counting (1, 2, 3) and a number that contains a fractional part (0.5, 3.14159, 4.2).

A data type is specified by a *reserved word*, a name that is predefined by MetaSL for use in MetaSL programs. For example, the reserved word `int` specifies values that are integers. Many of these data-type names are derived from widespread conventions in other programming languages. A typical example is the name `float`, an abbreviation for “floating-point number.” A `float` is a number that contains a decimal point. Within the precision of the hardware, a float can have any number of digits on either side of the decimal point, like 127.5 or 1.3333 — the decimal point can “float” to any position among the digits.

A data type that consists of a single type is called a *simple data type*. Collections of simple data types can define *complex data types*. For example, a color (in traditional computer graphics implementations) typically consists of red, green, and blue components. A color type can be defined as a complex type by specifying three values of type `float` for its red, green, and blue components.

2.1.2 Declaring a variable

A *variable* is a name defined in a MetaSL program that refers to a value of a certain data type. These three aspects — its name, data type, and value — are used in a *variable declaration* that creates the variable.

To *declare* a variable of a simple type, you write the data type and name of the variable, followed by its value. For example, to define a variable of type `int` named `year` with a value of 2010, you would write:

```
int year = 2010;
```

Because the integer 2010 is the initial value that the variable will have, declaring a variable with a value is called *initializing a variable*.

All simple variable declarations can follow this pattern:

```

      name
      ↓
int year = 2010;
  ↑       ↑
data type value

```

The value is preceded by the “equals” character (=) which associates the value with the variable. The declaration ends with the semicolon character (;) that serves the same purpose as a period at the end of a sentence.

```

assignment operator
      ↓
int year = 2010;
              ↑
            end of declaration

```

2.1.3 Declaring variables of complex data types

MetaSL defines a number of complex data types that are useful in rendering: `Color3` for colors, `float3` for a set of three `floats` used for mathematical concepts like points and vectors. You can think of a `Color3` type as having a single value that defines a color, or as a set of three values that define the red, green, and blue components. The `float3` is just one member of a family of types that are based on the simple type `float`; a `float2` contains two `float` values, a `float4` contains four.

To define a value for the `Color3` type, you can specify the red, green, and blue components of the color separated by commas in parentheses after the `Color3` data type:

```
Color3 light_blue = Color3 ( 0.8 , 0.9 , 1.0 );
                          ↑   ↑   ↑
                          red green blue
```

Rather than repeating the `Color3` data type, another structure, or *syntax*, for declaring complex types may also be used. In this syntax, the list of components directly follows the name of the variable:

```
Color3 light_blue ( 0.8 , 0.9 , 1.0 );
                  ↑   ↑   ↑
                  red green blue
```

In these examples of declarations, *space characters* are added to the declarations to make the syntax clearer. In a MetaSL program, space characters are optional in many cases; you would probably write something more compact:

```
Color3 light_blue(0.8, 0.9, 1.0);
```

This structure, in which a name is followed by values separated by commas and surrounded by parentheses, is the same as the syntax you will see for *functions* in a later section. A function calculates a value based on its *arguments*, the values in the parentheses. For this form of the declaration of a `Color3`, the red, green, and blue values in parentheses serve as the arguments to a function that creates a `Color3`. Because this structure constructs a new `Color3` variable, this form is called a *constructor* for the data type.

As a shortcut for constructors like `Color3` in which the same value should be used for all the components of the complex type, you can just specify that single value. In the following declarations, variables `gray_1` and `gray_2` have the same value: a `Color3` with 0.5 for all three components.

```
Color3 gray_1(0.5, 0.5, 0.5);
Color3 gray_2(0.5);
```

Some types are simply a different name, or *type synonym*, for a base MetaSL type. The `Color3` type is a different name for `float3` — both contain three floating point values, but `Color3` may give the reader of a program a better idea of that variable’s intended use. To create a synonym for a type, you use the *typedef* specifier with the name of an existing type and the synonym you are creating.

```
typedef float3 Point ;
```

The diagram illustrates the `typedef` declaration. It shows the code `typedef float3 Point ;`. An arrow labeled "existing type" points to the `float3` part of the code. Another arrow labeled "new synonym for float3" points to the `Point` part of the code.

2.1.4 Containing multiple values in array

The previous simple and complex data types all define a fixed number of components. An *array* is a complex data type that can contain an arbitrary number of values of the same type. The values that the array contains are called *array elements*.

To declare a variable to be an array of a certain type, you add the size of the array in square brackets after the name of the variable:

```
int primes [ 7 ] ;
```

The diagram illustrates the array declaration `int primes [7] ;`. It shows the code with labels and arrows: "array name" points to `primes`, "element type" points to `int`, "array size" points to `7`, and "square brackets" points to the `[` and `]` characters.

This declaration creates an array without any elements to be used later in a MetaSL program. To define an initial value for the array when it is declared, its value is specified using the “equals” character (=) in the same manner as for the types already described. You list the values to be contained by the array within *curly braces* ({}) and separated by commas.

The diagram shows the code `int primes[7] = { 2, 3, 5, 7, 11, 13, 17 };`. Annotations include:

- An arrow labeled "opening curly brace" pointing to the opening curly brace of the initialization list.
- An arrow labeled "closing curly brace" pointing to the closing curly brace of the initialization list.
- An arrow labeled "seven integers to initialize the array, separated by commas" pointing to the list of integers inside the braces.

An element in an array is described by its position, or *index*, with the first element having index zero, the second having index one, and so on. Rather than thinking of the index as a number like “first” or “second,” you should think of it as the “distance” from the first element in the array — there are no elements before the first element, so it is element zero; the second one is one element away from the first one, and so forth.

An array can contain any type of value, as long as all the values are of the same type. For example, an array can contain a series of values of type `Color3`. To initialize the array, you can use the constructor syntax for `Color3` to define the elements in the initialization list.

```
Color3 three_colors[3] = {
    Color3(0.0, 0.0, 0.0),
    Color3(1.0, 0.0, 0.0),
    Color3(1.0, 0.8, 0.0)
};
```

Declarations and their initial values can extend over more than one line to help clarify their structure. However, the structure of this array of `Color3` values is the same as an array of `int` values: a list in curly braces, with the values separated by commas.

In these examples of array, the *size* of the array — the number of elements it contains — is part of its declaration. Arrays may be declared without a size, an *unsized array*. This is useful when the size of the array is not known in advance, for example, when an array is passed as an argument to a function (described in a later section).

2.1.5 Literal values

In all of the examples above, the numeric values used in initializing variables were specified directly — 2010, 0.8, 0.9, 1.0 — and are therefore called *literal values*, or simply, *literals*. Typically, the initial values for variables will be literals. But once a variable has been declared, that variable can be used in the initialization of other variables. For example:

```
int year = 2010;
int next_year = year + 1;
```

The initial value of `next_year` in this declaration, `year + 1`, is an *expression*, which is the topic of the next section.

2.2 Expressions

Variables and literals can be combined into expressions using a subset of the C++ operators. Components of complex types can be specified using the name of the field and dot notation from C++ class instances. Arithmetic operations of variables of different simple types perform type promotion to the

type of higher precision of the operands. An expression with two operands of the same complex type performs pair-wise operations on the operand's components. Expressions containing complex types will perform promotions if possible; an operation on a simple type and a complex type will replicate the simple type to match the dimension of the complex type. Operator precedence is identical to C++.

2.2.1 Symbols in expressions

An *expression* is a value created by combining one or more variables and literals using combinations of symbols like `+`, `*`, and `/` that have meanings defined by MetaSL. Many of the meanings of these symbols are familiar from arithmetic, like the plus (`+`) character used for addition. Other symbols are based on standard conventions in other programming languages, like the asterisk (`*`) character for multiplication. Because these operators implement the basic calculations of arithmetic, they are called *arithmetic operators*. The variables and literals combined by operators are called *operands*. Most operations combine two expressions that are placed on either side of the operator. The expression on the left side of the operator is called the *left operand*; the expression on the right is called the *right operand*.

For example, if the variables `width` and `height` have already been declared and initialized, then you would use the multiplication symbol to calculate the area represented by the width and height of a rectangle. In the following expression, the `width` and `height` variables are the left and right operands of the multiplication operator.

```
width * height
      ↑
multiplication
```

The result of an arithmetic operation using two numbers is another number. A *Boolean expression* is a *comparison* of two numbers and produces a value of either `true` or `false`. For example, the character `<` means “less than,” so the expression `1 < 10` has a value of `true`, but the expression `3 < 2` has a value of `false`. The various *Boolean operators* in MetaSL are a standard set in most programming languages and have the same meaning as you would expect from inequalities in algebra. (The full set of MetaSL operators are listed by their category in the MetaSL Reference on page XX.)

More complex expressions can be defined by enclosing the sub-expressions to be calculated first in parentheses.

```
( width_1 * height_1 ) + ( width_2 * height_2 )
      ↑                   ↑
first sub-expression    second sub-expression
```

Without parentheses, arithmetic operations are performed in pairs in an order called *operator precedence*. For example, multiplication and division is calculated before addition and subtraction. Operator precedence means that the parentheses were not strictly necessary in the previous example; the two multiplications would have been performed first, and those two values then added together. However, adding parentheses is always a good idea if you are unsure of the precedence rules or would simply like to clarify the meaning of the expression. (See the precedence table in the MetaSL Reference on page XX.)

2.2.2 Expressions with complex data types

The simple types that compose a complex types can also be used in an expression. To use an array element as a value in an expression, you put the element's index in square brackets after the array name.

```
primes[0] + primes[7]
```

Using square brackets to reference an element is the same syntax as the array's size declaration.

```

array size declaration
↓
int primes[7] = { 2, 3, 5, 7, 11, 13, 17 };
                ↑           ↑
                primes[0]  primes[6]

```

Even if the array contains complex types, the array indexing is done in the same manner.

```

Color3 three_colors[3] = {
    Color3(0.0, 0.0, 0.0) , ← three_colors[0]
    Color3(1.0, 0.0, 0.0) , ← three_colors[1]
    Color3(1.0, 0.8, 0.0) ← three_colors[2]
};

```

Complex types like `Color3` and `float3` have predefined names for their components. For `Color3`, the red, green, and blue components are called `r`, `g`, and `b`; for `float3`, the components are called `x`, `y`, and `z`, to represent a point in three-dimensional space.

To refer to a component in an expression, you follow the variable name with a period character (`.`) and the name of the component. For example, the blue component of a `Color3` variable called `tint` would be `tint.b`. A component of a complex type specified in this manner can be used in an expression in the same way as a simple type. For example, the *luminance* of a color is a single value derived from the different sensitivities of the human eye to red, green, and blue. The MetaSL expression for luminance for a color named `tint` would be:

```

( tint.r * 0.2126 ) + ( tint.g * 0.7152 ) + ( tint.b * 0.0722 )
    ↑           ↑           ↑
red fraction (21.26%) green fraction (71.52%) blue fraction (7.22%)

```

The types of the values and variables in an expression must make sense for the operation being performed. For example, MetaSL also defines a `String` type for variables with a value containing text. Multiplying a `float` value by a `String` value does not have a reasonable interpretation, and is therefore not allowed. However, multiplying an `int` by a `float` is possible; the `int` is first *converted* to a value of type `float`, and then the two `float` values are multiplied.

For an expression with types for which conversion is defined, values are converted to the type with the *highest precision* of all the values in the expression. For integers and floating-point numbers, the increase in precision to that of a `float` is obvious; you can't express a number between two and three with an integer.

2.2.3 Operations with complex data types

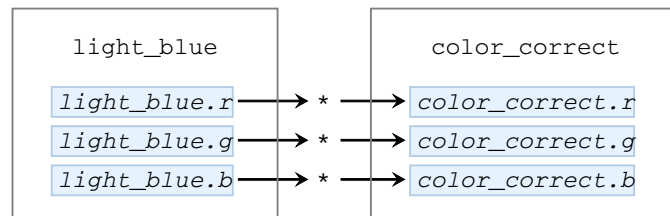
For two values with the same complex type, an arithmetic operation is the result of performing that operation on the separates components, matched by name. For example, if you have declared two `Color3` variables like this:

```
Color3 light_blue(0.8, 0.9, 1.0);
Color3 color_correct(0.8, 0.8, 0.5);
```

...then multiplying the two Color3 variables:

```
light_blue * color_correct
  ↑         ↑
Color3     Color3
```

...would result in three multiplications, one for each component:



Typically, if an arithmetic operation makes sense for the components of a complex type, then that operation will also be legal for the complex type itself.

2.2.4 Mixing simple and complex types in an expression

In the same way that a value of type `int` was converted to a `float` when it was multiplied by a `float`, simple types may be implicitly converted to a complex type in an expression. For example, multiplying a `Color3` value by a `float` value first converts the `float` value into a value of type `Color3` that uses the `float` value for all three of its components.

For example, if you had declared variables `darken` and `light_blue` in this way:

```
float darken = 0.5;
Color3 light_blue(0.8, 0.9, 1.0);
```

Then this expression in which you multiply the two variables:

```
darken * light_blue
  ↑       ↑
float   Color3
```

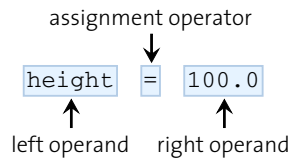
...is equivalent to multiplying the variables as if they had both been declared as `Color3` variables instead.

2.2.5 Assignment expressions

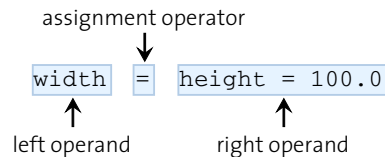
All of the previous expressions have consisted of literals and variables that are combined using operators that represent arithmetic and Boolean operations. In a MetaSL program, the value of an expression is the result of replacing all the variables with their values, and using those values as operands. Determining the value of an expression is called *evaluating* the expression. Evaluating a literal produces that literal value; evaluating

a variable produces that variable’s value; and evaluating two expressions combined with an operator is the result using that operator with the evaluations of the two operands.

However, by only using the operators describe so far, there is no way to retain the result of evaluating an expression. This is made possible by the *assignment operator*, represented by the equals character (=). The assignment operator redefines the value of a variable given as the left operand to be the value of the expression given as the right operand.



The value of the assignment operator is the value that is assigned to the left operand. Because assignment is an expression, it can be combined with other expressions. For example, two variables can be modified at the same time by using an assignment expression as the right operand of another assignment expression.



The assignment to variable `height` is made first, yielding the value 100.0, which is then assigned to variable `width`. When more than one assignment operator occurs like this in an expression, the evaluation is grouped to the right — the assignment operator is said to be *right-associative*.

The left operand for assignment must behave like a variable — it must be able to store a value. An expression that can store a value is called an *lvalue* — short for “left value” — because it occurs on the left side of the assignment operator. Conversely, the kind of expression that can occur on the right side of the assignment operator is called an *rvalue* — for “right value.” An lvalue is a more restricted type of expression than an rvalue; only variables and the components of variables can be lvalues, but any expression can be evaluated as a potential rvalue.

From these examples of the way that MetaSL uses the “equals” character you can also see that it has a different meaning than in algebra. In algebra, “=” means “is equal to”; in MetaSL, it means “now has the value of.” (This convention for the meaning of the “equals” character is used in many programming languages.)

Because of the different meaning that the “equals” character has in a programming language compared to algebra, an assignment expression that modifies an integer variable named `counter` in this way:

```
counter = counter + 1
```

...is meaningful in MetaSL — it replaces the previous value of `counter` with one greater than that value. The MetaSL expression cannot be read as if it were an algebraic expression, however; there is no possible value of `counter` for which the expression is true. But based upon the definition of the operands of the assignment operator, the difference with algebra becomes clearer: here the variable `counter` is being used both as an rvalue *and* as an lvalue.

This modification of the left operand in an assignment expression is the first example so far of a change to the value of a variable once it has been initialized. This change of state is called a *side effect* of the evaluation of the assignment expression. The side effects of expression evaluation are retained through a grouping structure in MetaSL called the *statement*, described in the next section.

2.3 Statements

Statements in MetaSL implement variable assignment, conditional execution, and iteration (for, while, and do/while) using C++ syntax. A sequence of statements enclosed by curly braces is treated as a single statement in the same contexts as in C++. MetaSL also provides additional iteration constructs using the foreach statement for light-emitting scene-data objects and for correlated sampling in one to four dimensions.

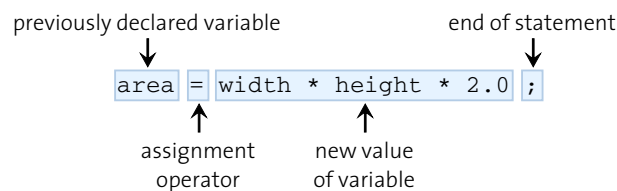
You can think of MetaSL's reserved words and operators, along with the variables you declare, as the vocabulary of MetaSL. *Statements* in MetaSL use that vocabulary to specify an action that a MetaSL program will perform when it is executed. Like the sentence in human languages, a statement is the smallest structure that can stand on its own. A MetaSL program consists of a sequence of statements; this section describes the various types of MetaSL statements.

2.3.1 Assigning a new value to a variable

In declaring a variable, you specify an initial value by following the variable name with an “equals” character (=) and the value:

```
float width = 8.5;
float height = 11.0;
float area = width * height;
```

The variable name, equals character, and value have the same structure as the assignment operation described in the previous section. However, the role of a declaration is to create a new variable and give it a value. The assignment operator *replaces* the current value of a variable with the value of the right operand. To create a statement from the assignment expression, you simply add the semicolon character to the end of the expression. As in a variable declaration, the semicolon functions like a period at the end of an English sentence.



Creating a statement from an assignment expression is one example of an *expression statement*. The syntax for an expression statement is quite simple — a semicolon is added to the end of an expression. However, unless the expression has a side effect, the execution of an expression statement will not produce a value that is available in other statements later in the program.

At this point, an expression statement that performs variable assignment may seem to duplicate the function of variable declarations. The usefulness of assignment expressions will become clearer in the more complex statements of the next sections.

2.3.2 Repeating statements

How can you calculate the sum of the seven prime numbers contained in the array `primes`? An obvious method is to create an expression in which all the elements are added together.

```
int primes[7] = { 2, 3, 5, 7, 11, 13, 17 };
int sum = primes[0] + primes[1] + primes[2] + primes[3] +
          primes[4] + primes[5] + primes[6];
```

This method is only convenient if the size of the array is small and is impossible if the number of elements in the array is not known in advance. If, however, the sum is calculated using seven statements in the following way, a pattern is displayed that suggests how the original expression can be simplified.

```
int primes[7] = { 2, 3, 5, 7, 11, 13, 17 };
int sum = 0;
sum = sum + primes[0];
sum = sum + primes[1];
sum = sum + primes[2];
sum = sum + primes[3];
sum = sum + primes[4];
sum = sum + primes[5];
sum = sum + primes[6];
```

In the seven statements that incrementally modify the variable `sum`, the only varying quantity is the array index. In MetaSL, the `for` *statement* can represent the repeated modifications to the variable `sum` in a single statement. A structure that implements repetition in programming language is often called a *loop*, so the `for` statement is also called a *for loop*. Because the `for` statement does not perform a single action but directs the flow of control of the program, a `for` statement implements *control flow* in MetaSL.

The `for` statement specifies a set of statements that should be repeated and defines the repetition: how variables used by those statements should be initialized, how they should be modified for each repetition, and when the repetition should stop. These defining components of the loop follow the reserved word `for` and are enclosed in parentheses. The statements to be repeated follow the defining components and are enclosed in curly braces. This group of statements is called the *body* of the loop.

```
for ( initialization ; ending test ; repeated action ) {
    loop body
}
```

For example, to add up the values of all seven elements of the array `primes`, the `for` statement should:

- Start with an index of zero;
- Stop when the index is bigger than six; and
- Use every integer from zero up to and including six as an index.

The initialization takes the form of a variable declaration. The ending test is a Boolean expression — the repetition stops when the expression has a value of `false`. The repeated action is a statement containing

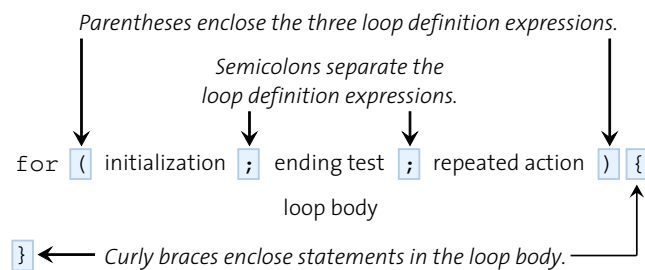
an assignment expression. Assuming that the variables `primes` and `sum` have declared as above, the `for` statement to calculate the total of the array elements contains a single statement that updates `sum` as the running total.

```

      initialization  ending test  repeated action
      ↓              ↓            ↓
for ( int i = 0 ; i < 7 ; i = i + 1 ) {
    sum = sum + primes[i];
}

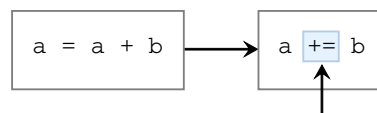
```

The structure of the `for` statement is defined by the position of the parentheses and curly braces. The semicolon separates the expressions that define the behavior of the repetition. Such characters, used to create boundaries within the code, are called *delimiters*.



2.3.3 Common assignment shortcuts

MetaSL provides shorter forms for several types of statements that contain assignment expressions. For example, modifying a variable by using its value in an operation has a shorter form in which the variable is not repeated.



Add the right-side value to the left-side value, then assign that new value to the left-side variable.

In the calculation of the sum of array elements, the `sum` variable is updated in this way.

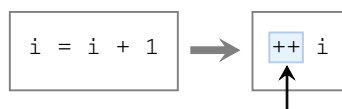
```

for (int i = 0; i < 7; i = i + 1) {
    sum += primes[i];
}

```

Add the array element value to the value of `sum`, then assign that new value to `sum`.

Adding a value of one to a variable is another typical operation. The briefer form eliminates the “equals” sign.



Add one to the value of the variable, then assign that new value to the variable.


The `++` syntax is useful for adding one to a variable that is used as an index into an array in the loop body.

Add one to the value of `i`, then assign that new value to `i`.

```

for (int i = 0; i < 7; ++i) {
    sum += primes[i];
}

```



2.3.4 The final form of the `for` statement example

MetaSL defines other methods for repeating a set of statements that will be described in examples developed in a later section. However, the structure of many other parts of MetaSL are represented in this small example: declarations, statements, delimiters, and the flow of control.

```

int primes[7] = { 2, 3, 5, 7, 11, 13, 17 };
int sum = 0;
for (int i = 0; i < 7; ++i) {
    sum += primes[i];
}

```

2.3.5 Choosing which statements to execute

The `for` loop uses a Boolean expression for a very specific purpose: to terminate the execution of the loop. The `if statement` uses a Boolean expression in a more general way: to decide whether or not a series of statements should be executed. Like delimiters in the `for` loop, the controlling information — the Boolean expression — is surrounded by parentheses, followed by a set of statements — the *clause* of the `if` statement — contained within curly braces.

```

if ( Boolean expression ) {
    clause to be executed if expression is true
}

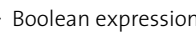
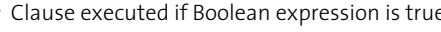
```

The Boolean expression and the clause of the `if` statement use variables declared before the `if` statement. If the value of the Boolean expression is `true`, then the statements in the clause are executed. In the following example, the two assignments to the red and green components of the `Color3` variable `background` are only executed if the value of the variable `scale` is greater than 1.0.

```

float scale = 2.0;
Color3 background(1.0, 0.0, 0.0);
if ( scale > 1.0 ) {
    background.r = 0.0;
    background.g = 1.0;
}

```

An `if` statement can be followed by another reserved word in MetaSL, `else`. After the `else` another set of statements are enclosed in curly braces. These statements are only executed if the value of the Boolean

expression is `false`, that is, if the first set of statements were *not* executed. These two groups of statements are often called separately the *if clause* and the *else clause*.

```

if ( Boolean expression ) {
    clause to be executed if expression is true
} else {
    clause to be executed if expression is false
}

```

In practice, the two clauses of the `if/else` form of the `if` statement may manipulate the same variable, depending upon the value of the Boolean expression.

```

float scale = 2.0;
Color3 background(1.0, 0.0, 0.0);
if ( scale > 1.0 ) {
    background.r = 0.0;
    background.g = 1.0;
} else {
    background.b = 0.5;
}

```

Boolean expression

← Clause executed if Boolean expression is true

← Clause executed if Boolean expression is false

These examples of the `if` statement are contrived to simplify their explanation; the value of the variable `scale` is already known, so why would you need an `if` statement dependent upon it? Using variables with values that can vary and can't be known in advance is what makes *functions* useful, described in the next section.

2.4 Functions

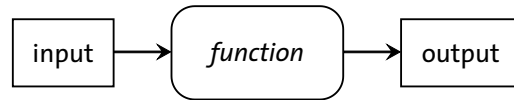
Functions are defined using standard C++ syntax for the return type, function name, the parameter list, and function body. However, passing values back from the function using pointers is not allowed. An extension to the C++ function syntax uses the keywords `in`, `out`, and `inout` to specify the copy semantics of function parameter passing, thereby implementing multiple return values from functions.

So far, the words used for the elements of MetaSL have been a mixture of terms and concepts borrowed from linguistics (“statement,” “declaration,” “clause”) and algebra (“expression,” “variable,” “Boolean”). Another concept from algebra that is fundamental to many programming languages is that of a *function*. In algebra, a function defines a relationship between numbers. For example, a function that produces even numbers can be written as follows:

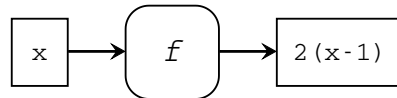
$$f(x) = 2(x-1)$$

The letter f represents a relationship that associates, or *maps*, any positive integer x to the positive integer $2(x-1)$. For example, the first even number is zero, the second is two, and the fifteenth is twenty-eight.

Another way of looking at a function that is useful in a programming language like MetaSL is that it represents a *process* in which an *input* is transformed into an *output*.



In this view, the function f takes one number as an input and produces another number as an output.

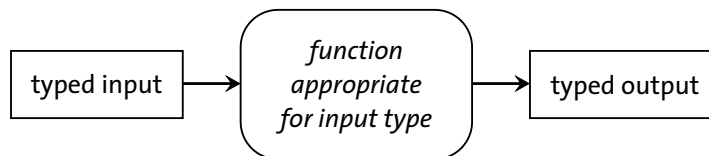


A function can be characterized by the type of input it requires and the type of output it will produce. Note that for this function to produce an even number, the input must be an integer greater than zero. With function f , any number — negative or with a fractional component — will produce an output value, but only with integers greater than zero does the function produce even numbers (that are integers by definition).

Other functions may be limited in the input values for which the function can produce an output value. For example, a function that divides one by its input value cannot have an output value for an input value of zero — dividing by zero is undefined.

$$g(x) = 1/x, \quad x \neq 0$$

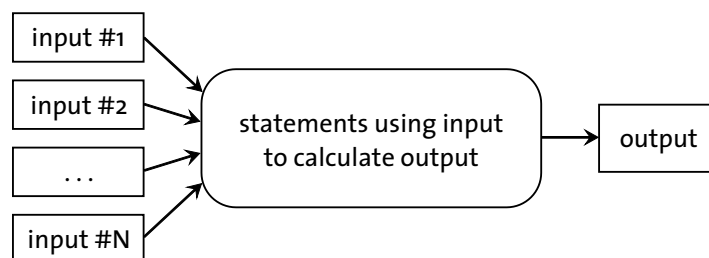
The restrictions on the input to a function corresponds to the limitations on the output a function can produce.



The definition of a function in MetaSL makes the kinds of input and output that are appropriate for the function part of the *function definition*. It describes what the function does using the variable declarations, statements, and delimiters already shown in the previous MetaSL examples.

2.4.1 Defining functions in MetaSL

To define a function in MetaSL, the concept of “input” will be generalized to include any number of input values.



In the MetaSL definition of a function, space characters and delimiters divide the function code into four parts:

1. The type of the output value, called the *return type*, because the value is said to be “returned” from the function
2. The name of the function
3. The input values, known collectively as the function’s *parameters*
4. The body of the function in which the statements are executed to calculate the *return value*

Parentheses are used to enclose the parameters; curly braces enclose the statements in the function body.

```

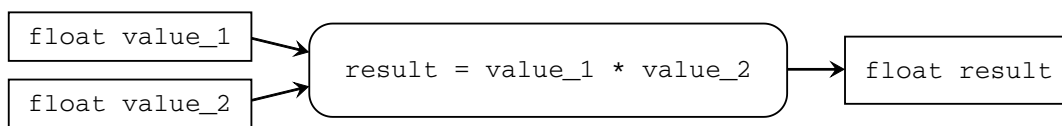
1. return type  2. function name  ( 3. parameters  ) {
                4. body
}

```

The function body can contain all the various structures of MetaSL for variable declaration, assignment, looping, and conditional execution. For example, a simple function that multiplies two input values defines its four parts as follows:

1. *Return type*: float
2. *Name*: multiply
3. *Parameters*: two values of type float
4. *Body*: statements that return the result of multiplying the parameters

The function can be displayed as a process with inputs and outputs.



The value “returned” by a function is defined with the *return statement* and follows the return reserved word in the function’s body.

```

1. return type  2. function name  3. parameters
   ↓           ↓                 ↓
float  multiply ( float value_1, float value_2 ) {
    float result = value_1 * value_2;
    return result;
}
                ↑
                4. body

```

The value defined by the `return` statement can be an arbitrary expression. The `multiply` function can be simplified by returning the multiplication of the parameters directly.

```
float multiply(float value_1, float value_2) {
    return value_1 * value_2;
}
```

Though the `multiply` function is trivial — it is just a lengthy replacement for the multiplication operator — it will be useful in showing how a function can be part of an expression.

2.4.2 Using a function in an expression

The parameters of the `multiply` function are the `float` variables `value_1` and `value_2`. When the function is defined, those variables do not have values; they are used symbolically in the statements in the function body. When the function is used in an expression — when you *call the function* — those symbolic variables are given actual values, called the function’s *arguments*, which are then used to calculate the function’s return value.

For example, the argument values can be floating point literals in an expression used to initialize a variable.

```

value used for argument value_1
      ↓
float area = multiply( 19.8 , 55.2 );
                        ↑
value used for argument value_2
```

Any expression that results in a value of the correct type can be used as an argument.

```

value of width is the value used for argument value_1
      ↓
float width = 19.8;
float height = 55.2;
float area = multiply( width , height );
                        ↑
value of height is the value used for argument value_2
```

The `multiply` function returns a value of type `float`. Because `multiply` also has parameters of that type, a call to `multiply` can be used as an argument to another call to `multiply`.

```

the return value from the multiply function is the value used for argument value_1
      ↓
float width = 19.8;
float height = 55.2;
float depth = 42.0;
float volume = multiply( multiply(width, height) , depth );
                        ↑
the value of depth is the value used for argument value_2
```

Here the distinction between “parameter” and “argument” helps explain what values the function can use when it is called. The function’s definition specifies the number and type of its parameters; when the function is called, the arguments — the actual values used in the function body — must match the parameters.

“Matching” in this case also includes the possibility of a conversion of an argument to the type required by the function; an `int` used for a `float` parameter would be converted to a `float` before it is used in the statements of the function. This conversion is permitted because the `float` type is typically of higher precision than the `int` type.

These examples show that there are two simple rules for calling a function:

1. The actual arguments can be any expression of the appropriate type, including a function call.
2. The function can be called wherever a value of the function’s return type may be used.

Only the return type and its parameters must be considered to determine if a function can be used in an expression. The contents of the function’s body — its *implementation* — is not relevant in this determination as long as the correct value is returned. Because of this, the function’s return type, name, and parameters fully characterize the function, and are therefore called its *signature*. Conversely, the implementation in the body is *encapsulated*, hidden from the rest of the program. Encapsulation allows you to improve the implementation of a function without regard to where the function is called as long as you do not change the function’s signature.

2.4.3 The MetaSL standard library functions

Functions that you write can contain all the MetaSL features described in the previous sections — variable declarations, statements, and flow-of-control structures like `if/else` and the `for` loop.

As part of the language, MetaSL also provides a set of pre-defined functions, called the *standard library functions*. These functions implement various mathematical operations, for example, finding the square root of a number or determining the sine and cosine of an angle. You can use these functions in the same way as the functions that you have defined.

Most of the standard library functions also can be called with arguments of various types, similar to the way that the addition operator (+) can be used with values of type `int`, `float`, `float3`, and so forth. A function that is defined for different types of arguments is said to be *overloaded*. The same kind of pair-wise calculation performed for operators will be done by functions that are defined in this way.

For example, if variables `p` and `q` have been declared like this:

```
float3 p = float3(0.3, 0.1, 0.9);
float3 q = float3(0.2, 0.5, 0.8);
```

...then the result of the `min` function using `p` and `q` as arguments:

```
float3 r = min(p, q);
```

...is exactly the same as calling the `min` function on each of the components and combining the result into a `float3`:

```
float3 r = float3(min(p.x, q.x),
                 min(p.y, q.y),
                 min(p.z, q.z));
```

In both cases, `r` is equal to `float3(0.2, 0.1, 0.8)`.

The full set of standard library functions is described in the MetaSL Reference on page XX.

2.4.4 Input and output function parameters

All of the simple functions described so far produce a single result and can therefore be part of any expression wherever a value of the function's return type can be used. However, it may be convenient to calculate and return more than one value from a function. A small addition to the syntax of the function's parameter list provides this feature.

Input parameters

In a function definition, a parameter is treated as a *local variable*, a variable that only has meaning within the body of the function.

```

                                function parameters
                                /      \
float multiply(float param_1, float param_2) {
    float result = param_1 * param_2;
    return result;
}

```

Function parameters are local variables in the function body.

By default, any modification of the parameters in the function body will not affect a variable that was supplied as an argument when the function was called.

For example, if the `multiply` function modified the `param_1` parameter, like this:

```

float multiply(float param_1, float param_2) {
    param_1 = param_1 * param_2;
    return param_1;
}

```

...then the value of a variable supplied for parameter `param_1` will not be modified, even though `param_1` is modified inside the function:

```

float var_1 = 2.0;
float var_2 = 3.0;
float var_3 = multiply(var_1, var_2);  Value of "var_3" is 6.0.
float var_4 = var_1;                  Value of "var_1" is 2.0 (unchanged).

```

MetaSL implements passing values to functions through parameters by *copying argument values*. The value of an argument is copied into the function as the value of the corresponding parameter in the function body. This is why the value of variable `var_1` in the previous example was not modified; once the value of `var_1` had been copied for use as the value of parameter `param_1`, the relationship between `var_1` and `param_1` was over.

This copy behavior is the default for function parameters. The parameter syntax in the previous function definitions is a shortcut for explicitly declaring the parameters to have this behavior using the *parameter qualifier* that means “copy the value to the parameter in the function.” The qualifier is the reserved word `in`, which precedes the parameter data type. A parameter qualified by `in` is called an *input parameter*.

```

                                explicit input parameter qualifiers
float multiply( in float param_1, in float param_2) {
    param_1 = param_1 * param_2;
    return param_1;
}

```

Output parameters

To modify a variable supplied as an argument in a function call, MetaSL provides another type of copy operation, defined with the parameter qualifier `out`. If a parameter in a function definition is preceded by `out`, then the value of the parameter at the end of the function body is copied back to the variable used for that parameter in the function call. A parameter qualified by `out` is called an *output parameter*.

For example, the following function, `add`, defines one output parameter and two input parameters. Because the resulting value of the function will be stored in the variable passed as the `result` parameter, the function itself will not return a value using the `return` statement. To declare that a function will not return a value, the reserved word `void` is used as the function's return type.

```

                                explicit input and output parameter qualifiers
no return value
void add( out float result, in float a, in float b) {
    result = a + b;
}

```

The copy qualifiers define the time at which the copying process occurs between the parameters of the function definition and the variables of the function call. The input parameters are copied before the statements of the function body begin execution; the output parameters are copied after the function body completes execution but before the function returns.

```

void add(out float result , in float a , in float b ) {
    result = a + b ;
}

```

Diagram illustrating the copying process:

- Arrows labeled "values copied in" point from the input parameters `a` and `b` in the function call to the corresponding parameters in the function definition.
- An arrow labeled "value copied out" points from the `result` parameter in the function definition back to the `result` parameter in the function call.

Because the first parameter of function `add` uses the `out` qualifier, calling `add` will modify a variable passed as the first argument.

```

float var_1 = 2.0;
float var_2 = 3.0;
float var_3 = 0.0;
add(var_3, var_1, var_2);
float var_4 = var_3;

```

Variable "var_3" must be initialized before it is used as an output parameter.
Value of parameter "result" in function "add" copied to "var_3".
Value of "var_4" is 6.0.

The argument passed to an output parameter must be able to have an assignment made to it – it must be an lvalue (as described in the section “Assignment expressions”). Because the value of an input parameter is simply copied into the function, no such restriction applies to it. Arguments supplied for input parameters can be any expression appropriate for the parameter type that can be assigned to a variable, including both

rvalues and lvalues — literals, variables, and compound expressions. When an lvalue is supplied as an argument for an input parameter, only its value is used; the assignment capability of an lvalue is not relevant in that case.

```
float var_1 = 0.0;      To be used as an output parameter, "var_1" must be initialized.
add(var_1, 2.0, 3.0);  Second and third arguments of "add" can be either lvalues or rvalues.
float var_2 = var_1;   Value of "var_2" is 6.0.
```

Input/output parameters

To define a function parameter that combines the copy behavior of both input and output parameter, the parameter type is preceded by the `inout` qualifier, defining an *input/output parameter*. A variable used for an input/output parameter has its value copied into the function, and can therefore be used in expressions in the statements of the function body. After the last statement of the function body, the current value is copied back to the variable, exactly like an output parameter.

```

                                inout/output qualifier
void offset( inout float result , in float amount ) {
    value copied out              values copied in
    result = result + amount ;
}

```

An input/output parameter allows a function to behave like the shortcut assignment operators (for example, `+=` and `*=`) in which the value of a variable is used to modify it (the variable is used both as an rvalue and as an lvalue).

```
float var_1 = 5.0;      To be used as an output parameter, "var_1" must be initialized.
offset(var_1, 1.0);     The new value of variable "var_1" is the sum of its value and 1.0.
float var_2 = var_1;   Value of "var_2" is 6.0.
```

Output parameter examples from standard library functions

Two of the standard library functions, `sincos` and `modf`, make use of output parameters. Function `sincos` calculates both the sine and cosine of an angle. (It may be more efficient for MetaSL to calculate them at the same time, and both may be useful in any case.) In the following example, the standard library function `radians` converts the angle measure in degrees to a measure in radians, the kind of measurement required by function `sincos`.

```
float sine_value = 0.0;
float cosine_value = 0.0;
float angle_in_degrees = 30.0;
float angle_in_radians = radians(angle_in_degrees);
sincos(angle_in_radians, sine_value, cosine_value);
```

No value is returned by `sincos` — it only uses two output parameters to return the sine and cosine of the angle provided as the first argument. No value is returned from `sincos`, so its return type is `void`; you can see this in the signature of `sincos`.

In contrast, the standard library function `modf` has one output parameter, but also returns a value. The `modf` function splits a number into two parts: the integral (whole number) part, and the fractional part.

By returning the fractional part as the value of the function, `modf` can be used in an expression. However, the integral part may also be useful, and can also be acquired in a single call to `modf` because of the output parameter.

```
float value = 3.14159;
float integral_part = 0.0;
float fractional_part = modf(value, integral_part);
```

Throughout the rest of the book, the standard library functions will play an important role in the implementation of the set of functions that define a *shader*, the topic of the next section.

2.5 Shaders

A shader in MetaSL is a plugin module for a rendering system, having the object-oriented conventions of member functions and variables. A shader defines zero or more input values and at least one output value. The primary method, called “main,” defines the output value of the shader.

A *shader* uses all the general programming structures described so far to define the primary structure in MetaSL specifically designed for rendering systems. The name “shader” is misleading, however. Though the first user-defined components of rendering systems were limited to determining the surface color (“shading”), the term is now widely used to describe a variety of user-customizable components of a rendering system. MetaSL shaders can customize other aspects of the rendering besides shading — the behavior of the virtual camera lens, the simulation of spotlights, the attenuation of light in a volume of smoke.

2.5.1 Shader syntax

The syntax of a shader resembles the declaration of a variable: the reserved word `shader` is followed by the shader’s name. However, the syntax of a shader definition also resembles the definition of a function; after the shader name, the shader body follows, surrounded by curly braces, and ending with a semicolon.

```
shader  shader name  {
      shader body
};
```

Within the shader body, sections of declarations and functions are separated by *labels*. The labels consist of the words “input,” “output,” or “member”, each followed by a colon character. The input and output section contain variable declarations. The member section contains variable declarations and functions specific to the shader. The variables declared in a shader are called *member variables*. The functions defined in a shader are called *methods*.

```

shader shader name {
  input:
    declarations of the input parameters
  output:
    declarations of the output parameters
  member:
    methods and member variables
};

```

The input section can contain values to be used by the shader in its calculations, called its *input parameters*, analogous to the parameters of a function. Each input parameter also specifies a *default value*, identical in form to the value supplied for a variable when it is initialized. The output section declares the variables that define one or more of the shader's result values, its *output parameters*. The member section must contain at a minimum a method called `main` which is responsible for calculating those outputs.

```

          shader name
          ↓
shader Constant_color {
  input:
    Color3 color = Color3(1.0, 1.0, 1.0); ← input parameter named "color"
  output:
    Color3 result; ← output parameter named "result"
  member:
    void main() {
      result = color; ← shader method named "main"
    }
};

```

The member section of this simple shader contains the single `main` method. The output calculation it performs is trivial — it simply provides the single input value as the shader's single *result*.

```

          1. function name
          ↓
2. return type  ↓  3. empty parameter list
void main () {
      result = color ;
    }
          4. output parameter      ↑
          5. input parameter      ↑

```

The main method has the typical structure of a function, with some restrictions on how it can be defined.

1. The name must be “main” — this is how MetaSL knows which method is responsible for calculating the result of the shader.
2. The main method does not return a result itself; so it has a return type of `void`, meaning “no return value.”
3. The input parameters of the shader are defined in the input section and are available in the shader body, so there are no method parameters defined for the main method itself.
4. Assigning a value to a variable declared in the output section defines the resulting value of the shader.
5. The input parameters can be used in the shader body, in this case, the single input parameter `color`. This result of this shader is the color provided as its single input.

There can be any number of input parameters. In the following shader, called `Blend`, two `Color3` parameters and one `float` parameter are used to blend between two colors, based on the weighting value defined by the float parameter called `color_1_fraction`.

```

shader name
  ↓
shader Blend {
  input:
  Color3 color_1 = Color3(0.0, 0.0, 0.0);
  Color3 color_2 = Color3(1.0, 1.0, 1.0); ← input parameters
  float color_1_fraction = 0.5;

  output:
  Color3 result; ← output parameter

  member:
  void main() {
    result = color_1 * color_1_fraction +
             color_2 * (1.0 - color_1_fraction); ← "main" shader method
  }
};

```

The `Blend` shader demonstrates all the fundamental parts of a shader: the input parameters, the output parameters, and the function that calculates the shader’s resulting value. The next chapter builds on this basis to show more complex shader techniques and the pictures they can create.