Color Transformation Language
User Guide and Reference Manual

Florian Kainz and Andrew Kunz

updated 06/04/2007

Contents

1 Introduction

1.1 Motivation

Digital color management requires translating digital images among different representations or color spaces. For example, the pixels in an image may encode the colors that should be seen when the image is displayed on a video monitor. Printing this image on paper or recording it on motion picture film requires transforming the pixels to an appropriate representation: Video, inks on paper and film all have different color gamuts and dynamic ranges. Color mixing is additive for video, but subtractive for inks and film. Video and film typically use three color channels, while four or more inks are used for printing on paper. A color management system must transform each pixel in the original image to corresponding amounts of ink or film density values.

The details of how each pixel is transformed can be fairly complex, and they are often subject to artistic decisions. When images are exchanged between different people or companies, it is often desirable to exchange exact descriptions of appropriate color transforms along with the digital image files. Two companies whose offices are in different geographical locations may each have a copy of the same digital image file. When one of the companies prints the image on paper, they want to be sure that they get the same result as the other company. In order to achieve identical results, the companies must agree on details of the printing process (for example, inks and paper), and they must agree on the transform that converts pixels in the file into amounts of ink on paper. Of course, this requires a description of the transform.

1.2 What CTL Is

The Color Transformation Language, or CTL, is a small programming language that has been designed to serve as a building block for digital color management systems. CTL allows users to describe color transforms in a concise and unambiguous way by expressing them as programs. Any digital color management system that supports CTL includes a CTL "interpreter", a software program that performs CTL-described operations on pixels that make up an image. In order to apply a given transform to an image, the color management system instructs the interpreter to load and run the CTL program that describes the transform. The original and the transformed image constitute the CTL program's input and output.

Color transforms can be shared by distributing CTL programs. Two parties with the same CTL program can apply the same transform to an image.

In addition to the original image, a CTL program can have input parameters whose settings affect how the input image will be transformed. For example, a transform may have an "exposure" parameter, such that changing the exposure makes the image brighter or darker. In order to guarantee identical results, parties that have agreed to use a particular transform must also agree on the settings for the transform's parameters.

General-purpose programming languages such as C, C++ or Python could of course be used to describe color transforms, but code written in those languages is not a suitable format for transform interchange. Some languages require the recipient to explicitly compile and link a program’s source code before the program can be executed. Code must be carefully written in order to make it portable across different operating systems. If code is executed inside a larger application, bugs can cause the application to malfunction. In addition, with most general-purpose programming languages, reliable protection from viruses and Trojan horses is very difficult to achieve.

A domain-specific programming language such as CTL can be designed to allow only the kinds of operations that are needed to describe color transforms. This improves the portability of programs, protects users against application software crashes and malicious code, and permits efficient interpreter implementations.

1.3 What CTL Is Not

A CTL interpreter is not a color management system; it is merely a software component that can be used as a part of such a system. The interpreter transforms images by executing CTL programs, but only when invoked by the rest of the system. The interpreter does not decide which programs to run and when, or what the settings for a transform's parameters should be.

CTL is a mechanism that can be used to implement any number of color management policies, but it does not by itself define a particular policy.

Also, CTL is not a general-purpose image processing language. CTL programs are restricted to performing color space transforms or other single-pixel operations. It is not possible to express operations such as convolving an image with a filter kernel, or computing the sum of all pixels in an image.

1.4 This Document

This document describes the CTL language and the reference implementation of the CTL interpreter. Section 2 provides an overview of the language. It should give the reader enough background to start writing CTL programs and understand their use. Section 3 describes the reference implementation of the interpreter for the language for readers who are integrating the reference CTL interpreter into their C++ applications. Section 4 provides more information on the implementation and Section 5 provides a programming reference guide for CTL.

This document assumes that the reader is familiar with the C and C++ programming languages and with basic digital imaging concepts.

The reference implementation is designed so that it is easy to interpret CTL properly using the provided parser and framework in combination with a custom program execution engine. Implementers who choose to re-implement the entire interpreter should produce the same behavior and results as the reference interpreter. In the event that discrepancies are found between the language reference in Section 6 and the reference implementation, the reference implementation should be considered the specification.

2 Overview of CTL

In order to make CTL programs look familiar to people who are used to programming in C or C++, the syntax of CTL was deliberately made similar to C. However, there are significant differences. Below, the main features of CTL are introduced by showing a number of code samples.

2.1 Basic Concepts

The first example is a CTL transform that adjusts the overall brightness of an image according to an "exposure" parameter:

```
void
adjustExposure
   (output varying half rOut,
    output varying half gOut,
    output varying half bOut,
    input varying half r,
    input varying half g,
    input varying half b,
    input uniform float e = 0)
{
    float f = pow (2, e);
    rOut = r * f;
    gOut = g * f;
    bOut = b * f;
}
```

In CTL, a transform is expressed as a function that describes the operations that must be performed on an individual pixel. In order to apply the transform to an image, the CTL interpreter calls the function once for each pixel.

The `adjustExposure()` function expects as inputs an image with three channels, `r`, `g`, and `b`, and an exposure value, `e`. The function multiplies all pixels by 2^e, thus making the image brighter or darker. (If the data in the pixels represent linear scene luminance values, then this operation is equivalent to changing the aperture of the iris on the camera's lens by `e` f-stops.) The result is a new image with three channels, `rOut`, `gOut` and `bOut`.

Most CTL functions have parameters. Input parameters supply input data, and results are returned via output parameters or via the function's return value.

Function `adjustExposure()` has four input parameters, `r`, `g`, `b` and `e`. The parameters are marked with the keyword `input`. The `varying` keyword indicates that the value of `r`, `g` and `b` varies from pixel to pixel. In other words, `r`, `g` and `b` are image channels. The keyword `uniform` indicates that the fourth parameter, `e`, is the same for all pixels.

In addition to its input parameters, `adjustExposure()` has three output parameters, `rOut`, `bOut` and `gOut`, marked with the keyword `output`. The `varying` keyword indicates that their values may vary from pixel to pixel; `rOut`, `bOut` and `gOut` are the channels of the function's output image.

The return type of `adjustExposure()` is `void`. As in C, a `void` return type indicates that the function does not return a value; all results are delivered via output parameters.

All function parameters and variables are typed. Here, parameter `e` as well as variable `f` in the function's body are of type `float`; that is, the values of `e` and `f` are 32-bit floating-point numbers. The other six parameters are of type `half`; their values are "half-precision" or 16-bit floating-point numbers.

In addition to floating-point numbers, CTL supports Boolean values, signed and unsigned integers, structures and arrays. Those data types are described below.

Each input parameter may optionally have a default value. When a function is called and the caller does not specify a value for a given parameter, then the called function uses the default value instead. The caller of `adjustExposure()` is allowed to omit the value of `e`. In this case `adjustExposure()` assumes that `e` is zero. Parameters `r`, `g` and `b` have no default values; the caller must explicitly specify their values.

The body of function `adjustExposure()`, between the curly braces, `{` and `}`, describes what the function does: first, built-in function `pow()` is called to compute 2^e, and the result is assigned to variable `f`. Then `r`, `g`, and `b` are multiplied by `f`, and the result is assigned to `rOut`, `gOut` and `bOut`, respectively.

All CTL functions operate on one pixel at a time; there is no explicit loop over the pixels in an image. When the CTL interpreter applies the transform that is described by function `adjustExposure()` to an image, it calls the function once for each pixel, with the `r`, `g` and `b` parameters set equal to the pixel's r, g and b values.

The order in which the per-pixel calls to a CTL function are executed is not observable from the point of view of the function. Calls may run one after another or multiple calls may run concurrently. Concurrent calls may really run simultaneously in multiple threads, or they may be interleaved in a single thread.

Note that a CTL function sees only a single pixel at a time. The function cannot access neighboring pixels. It cannot determine the size of the overall image or the location of the current pixel within the image. Limiting the view of the world to a single pixel does not restrict the implementation of pure color space transforms, where what happens to a given pixel does not depend on any other pixels. On the other hand, the single-pixel world view makes it impossible to express any operations that depend on neighboring pixels or on the current pixel's location, for example convolving an image with a blur kernel or generating color gradients.

Giving a CTL program access to only a single pixel at a time guarantees that each output pixel is strictly a function of the corresponding input pixel and data passed from the application to the CTL program. Neither state accumulated during previous program executions nor additional input pixels can influence the CTL program's output. This allows CTL interpreters to optimize execution in ways that would not be possible with a more general programming model.

2.2 Input and Output, Varying and Uniform

The `input` and `uniform` keywords in the declaration of function parameters are optional. If neither `input` nor `output` is specified for a parameter, then the parameter is an input parameter. If neither `varying` nor `uniform` is specified, then the parameter is assumed to be uniform. Function `adjustExposure()`, above, could have been declared like this:

```
void
adjustExposure
    (output varying half rOut,
     output varying half gOut,
     output varying half bOut,
     varying half r,
     varying half g,
     varying half b,
     float e = 0)
{
    ...
}
```

Why are the `varying` and `uniform` keywords necessary? Since each function operates on one pixel at a time, the body of a function does not depend at all on whether the function's parameters vary between pixels or not. A function such as

```
float
square (float x)
{
    return x * x;
}
```

should work exactly the same way for uniform values as for varying ones.

`varying` and `uniform` are hints to the color management system that calls CTL functions. The `varying` keyword tells the system that the corresponding function parameter is an image channel; the `uniform` keyword means that the parameter is something else, for example an "attribute" or "tag" from an image file's header.

The CTL interpreter makes the `uniform` and `varying` hints available to the color management system, but it ignores those hints otherwise. When the `square()` function, above, is called, it does not matter whether the value of x varies from pixel to pixel. Only CTL functions that are meant to be called directly by the color management system must specify which parameters are uniform and which are varying. Functions that will only be called by other CTL functions don't need to do this.

2.3 Data Types

CTL has the following six fundamental data types:

`bool`	Boolean, can hold the value `true` or `false`

`bool`	Boolean, can hold the value `true` or `false`
`int`	signed 32-bit integer
`unsigned int`	unsigned 32-bit integer
`float`	single-precision (32-bit) floating-point
`half`	half-precision (16-bit) floating-point
`void`	indicates that a function does not return a value

CTL supports two kinds of aggregate data types: arrays and structures. The elements of arrays and the members of structures can be fundamental or aggregate types. Structures and arrays can be nested to form arrays of structures, structures containing arrays and other structures, or multidimensional arrays. For example:

```
bool x[3];        // an array of 3 Booleans

int y[3][4][2];   // an array of 3 arrays of 4 arrays of 2 integers,
                  // in other words, a three-dimensional array

struct S          // a simple structure
{
    int x;
    int y;
};

struct T          // a nested structure, containing an integer
{                 // and a two-dimensional array of S
    int i;
    S   s[4][7];
};
```

Structures and arrays can be initialized by listing the values of their members or elements between curly braces:

```
float f[3] = {0.0, sqrt(2), 2 * sqrt(2)};
S s = {3, 4};
S t[2] = {{1, 2}, {3+4, 6/3}};
```

Except for function parameters, all array sizes are static. The CTL interpreter must be able to determine the size of an array at "compile time", when it parses the CTL source code. CTL does not support arrays whose size is determined at run-time.

The following array declarations are valid:

```
void
f ()
{
    const int n = 2;
    const bool i[n*2] = {true, false, false, true};
    const int j[][] = {{1, 2}, {3, 4}, {5, 6}};
}
```

The CTL interpreter can determine the size of array `i` because it can determine that the value of `n*2` is 4 without actually calling function `f()`. Similarly, by counting the values between the curly braces, the interpreter can determine that `j` is an array with 3 by 2 elements.

However, the following array declaration is not allowed:

```
void
g (int x)
{
    int k[x]; // error: cannot determine size of array k
}
```

In order to compute the size of array `k`, function `g()` would actually have to be called, and the size of `k` could be different for each call.

2.4 Arrays, Static Data, Lookup Tables

Many color transforms rely on lookup tables. In CTL, lookup tables can be represented as multi-dimensional arrays, as shown in the following example:

```
const float lut1D[] =
{
    0.03, 0.05, 0.10, 0.12, 0.15,
    0.20, 0.50, 1.00, 2.10, 3.30,
    4.45, 5.20, 5.40, 5.50, 5.50
};

const float lut1DMin = 0.0;
const float lut1DMax = 1.0;

const float lut3D[4][4][4][3] =
    {{{{.00, .00, .00}, {.11, .00, .00}, {.41, .00, .00}, {.91, .00, .00}},
      {{.00, .10, .00}, {.12, .20, .00}, {.42, .31, .00}, {.91, .49, .00}},
      ... // 45 lines omitted
      {{.00, .49, .53}, {.13, .60, .64}, {.43, .71, .75}, {.91, .93, .99}}}};

const float lut3DMin[3] = {0.00, 0.00, 0.00};
const float lut3DMax[3] = {5.50, 5.50, 5.50};

void
applyLuts
  (output varying half rOut,
   output varying half gOut,
   output varying half bOut,
   varying half r,
   varying half g,
   varying half b,
{
    half r2 = lookup1D (lut1D, lut1DMin, lut1Dmax, r);
    half g2 = lookup1D (lut1D, lut1DMin, lut1Dmax, g);
    half b2 = lookup1D (lut1D, lut1DMin, lut1Dmax, b);
    lookup3D_h (lut3D, lut3DMin, lut3Dmax, rOut, gOut, bOut, r1, g1, b1);
}
```

Function `applyLuts()` transforms three input values, `r`, `g` and `b`, into three output values, `rOut`, `gOut` and `bOut`, by applying a one-dimensional (1D) and a three-dimensional (3D) lookup table.

The lookup tables, `lut1D` and `lut3D`, are represented as two arrays of `float`; `lut1D` is one-dimensional and `lut3D` is four-dimensional (`lut3D` is a 3D table with entries that are 1D arrays). Both arrays are defined outside of function `applyLuts()`. The arrays are "static"; they are created and initialized once, when the program is loaded, rather than repeatedly, every time function `applyLuts()` is called.

In CTL, the keyword `const` must be used to mark all static data as constant. Constant static data are initialized when the program module that defines them is loaded. Static data cannot be modified after initialization. Non-constant static data are not allowed.

Prohibiting non-constant static data ensures that CTL functions cannot have side effects other than modification of the function's output parameters. This way, independent CTL function calls cannot exchange data via static variables, and the function calls can be executed sequentially, interleaved or concurrently, without affecting each call's results.

Function `lookup1D()` performs linearly interpolated 1D table lookups. `lookup1D()` is built into the CTL interpreter, but an equivalent function could also be written in CTL:

```
float
myLookup1D (float lut[], float xMin, float xMax, float x)
{
    unsigned int iMax = lut.size - 1;

    if (x1 > xMin && x < xMax)
    {
        float u = (x - xMin) / (xMax - xMin) * iMax;
        unsigned int i = u;
```

```
            u = u - i;

            return lut[i] * (1 - u) + lut[i + 1] * u;
        }
        else if (x >= xMax)
        {
            return lut[iMax];
        }
        else // either x <= xMin or x is a NaN
        {
            return lut[0];
        }
    }
```

The function's lookup table parameter, `lut`, is a "variable-size array", that is, an array of unspecified size. The number of elements in `lut` is not known until function `myLookup1D()` is called, and different callers may pass arrays of different sizes. Within `myLookup1D()`, the size of `lut` can be queried using the `size` operator. The expression

```
lut.size
```

returns the number of elements in the array.

The 3D table lookup function is also built into the interpreter, but again, an equivalent function could be written in CTL. Here we show only the function's name and parameter list, omitting the function's body:

```
void
myLookup3D
    (float lut[][][][3],
     float xMin[3],
     float xMax[3],
     float x[3],
     output float y[3])
{
    ...
}
```

Parameters `x`, `xMin`, `xMax` and `y` are fixed-size arrays, but the size of the `lut` parameter is only partially specified: `lut` is an array of `r` by `s` by `t` by 3 elements, where `r`, `s` and `t` are not known until `myLookup3D()` is called. Again, the size of `lut` can be queried using the `size` operator:

```
unsigned int r = lut.size;
unsigned int s = lut[0].size;
unsigned int t = lut[0][0].size;
unsigned int u = lut[0][0][0].size;  // returns 3
```

The table can be indexed with three or four indices, yielding either a 1D array or an individual floating-point number. For example:

```
float a[3] = lut[i][j][k];
float b = lut[i][j][k][h];
```

2.5 Scattered Data Interpolation

Using 3D lookup tables requires that the data in the table are known at 3D locations that form a regular grid. Tables are often generated from measured data where the measured points may not form a regular grid. In this case we must use scattered data interpolation to generate a regular grid from the measurements. We can do this by calling the built-in function `scatteredDataToGrid3D()`. Written in CTL, the signature of this function looks like this:

```
void
scatteredDataToGrid3D
    (float scatteredData[][2][3],
     float gridMin[3],
     float gridMax[3],
     output float grid[][][][3])
```

Function parameter `scatteredData` is an array of pairs of 3D points. The first element in each pair represents a location in 3D space where a 3D value, represented by the second element in the pair, was measured. Function `scatteredDataToGrid3D()` constructs a smooth function that interpolates the measured data, and then samples this function at regular intervals to form a 3D grid, `grid`. `gridMin` and `gridMax` indicate the grid's bounding box. To generate a static 3D grid from a scattered data set, we can call `scatteredDataToGrid3D()` like this:

```
const float data[][2][3] =
{
    {{0.0, 0.0, 0.0}, {0.0, 0.0, 0.0}},
    {{0.0, 0.2, 0.0}, {0.1, 0.3, 0.1}},
    ...
    {{2.0, 2.0, 2.0}, {2.1, 2.1, 2.2}}
};

const float gridMin[3] = {0.0, 0.0, 0.0};
const float gridMax[3] = {2.0, 3.0, 4.0};

const float grid[16][16][16][3],
    scatteredDataToGrid3D (data, gridMin, gridMax, grid);
```

Note the syntax for the initialization of array `grid`. The comma after the declaration of `grid` indicates that the following expression initializes the array. Within the initializing expression, the array is treated as variable, even though it has been declared `const`. This allows us to pass the array to function `scatteredDataToGrid3D()` as an output parameter.

The unusual initialization syntax is required because CTL does not support variable-size arrays except in function parameters. We could write a version of `scatteredDataToGrid3D()` that returns a grid with 16 by 16 by 16 points, but that function could not be used to initialize, for example, a grid with 64 by 64 by 64 points. In order to allow a function to initialize a grid of arbitrary size, the grid must be a variable-size array output parameter.

2.6 Standard Library

CTL comes with a standard library of built-in functions. Most of these functions could be written in CTL, but of course writing color transforms is easier if commonly used functions already exist. The standard library includes:

- elementary functions, such as `sqrt()`, `pow()`, `log()`, `sin()`, etc., with the same names and functionality as in C,
- floating-point classification functions to determine if a value is finite, infinite or not a number,
- 1D and 3D table lookups with linear interpolation, 3D scattered data interpolation,
- 3D vector and 4 by 4 matrix operations, such as addition, dot product, vector-times-matrix and matrix-times-matrix multiplication, and
- conversions among a few standard color spaces: RGB with arbitrary primaries and white point, XYZ, L*u*v* and L*a*b*.

2.7 Modules

CTL programs may be split into multiple source files or "modules". This is useful for assembling libraries of commonly-used functions and lookup tables. For example, a `logLookup` module might contain a function that performs table lookups in a logarithmic space:

```
float
logLookup1D (float lut[], float xMin, float xMax, float x)
{
    return pow (10, lookup1D (lut, log10 (xMin), log10 (xMax), log10 (x)));
}
```

The CTL code for the module is stored in a file called `logLookup.ctl`, in a directory or folder where the CTL interpreter can find it.

Another module, `logLut`, may define a lookup table. This module is stored in file `logLut.ctl`:

```
const float logLut[] =
{
    0.03, 0.05, 0.10, 0.12, 0.15,
    0.20, 0.50, 1.00, 2.10, 3.30,
```

```
    4.45, 5.20, 5.40, 5.50, 5.50
};

const float logLutMin = 0.0;
const float logLutMax = 3.0;
```

Other CTL modules that want to call function `logLookup1D()` or use the `logLut` table can now "import" the `logLookup` and `logLut` modules. For example, module `applyLogLut`, stored in file `applyLogLut.ctl`, may contain the following:

```
import "logLookup";
import "logLut";

void
applyLogLut
    (output varying half r1,
     output varying half g1,
     output varying half b1,
     varying half r,
     varying half g,
     varying half b,
{
    r1 = logLookup1D (logLut, logLutMin, logLutmax, r);
    g1 = logLookup1D (logLut, logLutMin, logLutmax, g);
    b1 = logLookup1D (logLut, logLutMin, logLutmax, b);
}
```

When module `applyLogLut` is loaded, the `import` statements cause the CTL interpreter to also load modules `logLookup` and `logLut`. Function `applyLogLut()` can now call function `lookLookup1D()` and access array `logLut` as well as constants `logLutMin` and `logLutMax`.

3 Description of the CTL interpreter

The CTL interpreter is implemented as a set of C++ libraries that can be linked into a color management system or into any other application program that needs to run CTL programs. The interpreter's C++ programming interface allows applications to load CTL modules and to call the functions that are defined in those modules.

3.1 Theory of Operation

The CTL interpreter is split into a "front" and a "back" end. When a CTL module is loaded, the front end parses the module's source text and generates an abstract syntax tree, reporting syntactic and semantic errors in the process. The front end also maintains the interpreter's symbol table and performs constant expression evaluation and dead code elimination. For example,

```
const int n = 1;
foo (x + 4 / (n + 1));
```

becomes

```
foo (x + 2);
```

while the statement

```
if (1 > 2)
    bar();
```

is deleted entirely.

The interpreter's back end handles converting the abstract syntax tree into executable code and actually running the code. The interpreter's front end is constructed in such a way that multiple different back end implementations can be supported.

The reference CTL interpreter implementation includes a single back end. This back end generates instructions for a single-instruction-multiple-data (SIMD) virtual machine. This SIMD back end is portable, allowing CTL programs to be run on any platform that supports C++. Other, higher-performance back ends can be implemented for specific applications or hardware platforms. For example, the syntax tree produced by the front end could be compiled into native machine code. On platforms with GPU (graphics processing unit) support, the back end could generate OpenGL shading language, HLSL or Cg code and run it on the GPU.

The SIMD back end is fast enough for still images of moderate resolution, but working with high-resolution images or real-time playback of moving images requires either a faster interpreter back end or other acceleration techniques (see section 3.9).

3.2 C++ API

This section gives a brief overview of the CTL interpreter's C++ application programming interface (API). Calling the API shown here is somewhat cumbersome because of the complex mechanism for passing function call arguments and return values back and forth between CTL and C++. In many cases, writing an abstraction layer on top of the interpreter's API, with a simplified, application-specific argument passing mechanism, will be more convenient than calling the interpreter directly. One such abstraction layer, which simplifies processing of images that are stored in OpenEXR format, is described in Appendix A.

In C++, the CTL interpreter front end is implemented in class `Ctl::Interpreter`. Class `Interpreter` is an abstract base class. The back end adds a concrete class `SimdInterpreter` that is derived from class `Interpreter`. Once we have an interpreter, we can load CTL modules, and we can call CTL functions.

The following example creates an interpreter with a SIMD back end, and loads a CTL module:

```
SimdInterpreter interp;
interp.loadModule ("adjustExposure");
```

Loading the module proceeds as follows:

The environment variable `CTL_MODULE_PATH` is interpreted as a colon-separated list of directory names. The interpreter visits each of the directories listed in `CTL_MODULE_PATH` and looks for a file called `adjustExposure.ctl`. For instance, if `CTL_MODULE_PATH` contains the string

```
ctl:/home/ctl:/usr/local/ctl
```

then the CTL interpreter checks if any of the following files exist:

```
ctl/adjustExposure.ctl
/home/ctl/adjustExposure.ctl
/usr/local/ctl/adjustExposure.ctl
```

The CTL interpreter opens the first `adjustExposure.ctl` file it finds. The Interpreter parses the contents of the file and generates executable SIMD code. If file `adjustExposure.ctl` contains any `import` statements, then the modules named in the import statements are loaded as well. If any file cannot be found or if errors are found during parsing, then `loadModule()` throws an exception. Otherwise, `loadModule()` returns, and we are ready to call the functions defined in module `adjustExposure`.

We assume that the module we have just loaded defines an `adjustExposure()` function. In order to call this function, we must create a `FunctionCall` object:

```
FunctionCallPtr call = interp.newFunctionCall ("adjustExposure");
```

The `FunctionCallPtr`, above, is a reference-counting pointer to the `FunctionCall` object. If the pointer goes out of scope and there are no other `FunctionCallPtrs` that point to the same `FunctionCall` object, then the `FunctionCall` object is automatically deleted.

The `FunctionCall` object has methods to access the arguments and the return value of function `adjustExposure()`. The argument and the return value are represented as `FunctionArg` objects. Each `FunctionArg` object has a pointer to a `DataType` object which describes the argument's type. The `FunctionArg` object also has a `data` buffer for the argument's value.

Before calling a CTL function, the C++ application must set the function's input arguments. After the CTL function returns, the application can examine the output arguments and the return value. We assume that CTL function `adjustExposure()` has the following signature:

```
void
adjustExposure
    (output varying half rOut,
     output varying half gOut,
     output varying half bOut,
     varying half r,
     varying half g,
     varying half b,
     float e = 0)
```

In order to simplify the code below, we assume that the C++ application that calls `adjustExposure()` is written so that it knows the parameter list in advance, instead of discovering the parameters at run time.

Given a pointer to a `FunctionCall` object that refers to the CTL function `adjustExposure()`, as well as a value for `e`, and buffers for `n` pixels worth of image data (input channels `r`, `g`, and `b`; and output channels `rOut`, `rOut` and `bOut`), the C++ function `callCtlChunk()` passes the input pixel data to the CTL interpreter, calls the CTL function, and retrieves the output pixel data from the CTL interpreter:

```
void
callCtlChunk
    (FunctionCallPtr call,
     size_t n,
     half rOut[],
     half rOut[],
     half rOut[],
     const half r[],
     const half g[],
     const half b[],
     float e)
{
    // First set the input arguments for the function call:

    FunctionArgPtr rArg = call->findInputArg ("r");

    if (!rArg ||
        !rArg->type().cast<HalfType>() ||
        !rArg->isVarying())
```

```
    {
        // The CTL function has no argument "r", the argument
        // is not of type half, or the argument is not varying

        throw ArgExc ("Cannot set value of argument 'r'");
    }
    else
    {
        memcpy (rArg->data(), r, n * sizeof (half));
    }


    FunctionArgPtr gArg = call->findInputArg ("g");
    ...

    FunctionArgPtr bArg = call->findInputArg ("b");
    ...

    FunctionArgPtr eArg = call->findInputArg ("e");

    if (!eArg ||
        !eArg->type().cast<FloatType>() ||
         eArg->isVarying())
    {
        // The CTL function has no argument "e", the argument
        // is not of type float, or the argument is not uniform

        throw ArgExc ("Cannot set value of argument 'e'");
    }
    else
    {
        *(float*)eArg->data() = e;
    }

    // Now we can call the CTL function for
    // pixels 0, through n-1

    call->callFunction (n);

    // Retrieve the results

    FunctionArgPtr rOutArg = call->findOutputArg ("rOut");

    if (!rOutArg ||
        !rOutArg->type().cast<HalfType>() ||
        !rOutArg->isVarying())
    {
        // The CTL function has no argument "rOut", the argument
        // is not of type half, or the argument is not varying

        throw ArgExc ("Cannot set value of argument 'rOut'");
    }
    else
    {
        memcpy (rOut, rOutArg->data(), n * sizeof (half));
    }

    FunctionArgPtr gOutArg = call->findOutputArg ("gOut");
    ...

    FunctionArgPtr bOutArg = call->findOutputArg ("bOut");
    ...
}
```

The code above uses the functions `FunctionCall::findInputArg()` and `FunctionCall::findOutputArg()` to lookup parameters by name: `findInputArg()` and `findOutputArg()` return either a pointer to the `FunctionArg` that corresponds to the CTL function parameter with the specified name, or 0 if the CTL function has no such parameter.

As written above, `callCtlChunk()` can handle only a limited number of pixels at a time. This is because the maximum value of `n` that can be passed to `call->callFunction()` is limited by the implementation of the CTL interpreter back end. (For the reference SIMD back end, the limit is a few thousand pixels.) In order to handle images of arbitrary size, the pixel data must be broken up into smaller chunks:

```
void
callCtl
    (Interpreter &interp,
     FunctionCallPtr call,
     size_t n,
     half rOut[],
     half rOut[],
     half rOut[],
     const half r[],
     const half g[],
     const half b[],
     float e)
{
    while (n > 0)
    {
        size_t m = min (n, interp.maxSamples());
        callCtlChunk (call, m, rOut, gOut, bOut, r, g, b, e);

        n    -= m;
        rOut += m;
        gOut += m;
        bOut += m;
        r    += m;
        g    += m;
        b    += m;
    }
}
```

3.3 FunctionCall and FunctionArg Objects

The code shown above already knows the signature of the CTL function it intends to call. Sometimes a C++ application program must call a CTL function whose signature is not known in advance. In this case, the application must query the corresponding `FunctionCall` object in order to discover the names and types of the function's parameters.

Class `FunctionCall` has the following member functions:

`returnValue()`	Returns a `FunctionArgPtr` that points to the CTL function's return value.
`numInputArgs()` `numOutputArgs()`	Returns the number of input or output arguments.
`inputArg(i)` `outputArg(i)`	Returns a `FunctionArgPtr` that points to the input or output argument number `i`. Input arguments are numbered from `0` to `numInputArgs()-1`; output arguments are numbered from `0` to `numOutputArgs()-1`. The numbering of the arguments does not necessarily correspond to the order of the function's parameters in the CTL source code.
`findInputArg(n)` `findOutputArg(n)`	These functions return a pointer to the input or output argument with name `n`, or a null `FunctionArgPtr` if the function has no argument with name `n`.

Given a pointer to a `FunctionArg`, an application program can call its member functions to get more information:

`name()`	Returns the `FunctionArg's` name. This is the same as the name of the function's parameter in the CTL source code.

`type()`	Returns a `DataTypePtr` that points to a `DataType` object, which describes the `FunctionArg`'s type. `DataType` objects are described in section 3.4.
`isVarying()`	Indicates if the `FunctionArg` is varying or uniform.
`data()`	Returns a `void` pointer that points to a buffer for the argument's value. Before calling a CTL function, the application program must store values for the function's input arguments in the corresponding buffers. After calling the CTL function, the application can read the buffers for the output argument values and the return value.
`hasDefaultValue()`	Returns `true` if the `FunctionArg` has a default value. Calling `setDefaultValue()` makes the `FunctionArg` equal to the default value. The application program can read a `FunctionArg`'s default value by first calling `setDefaultValue()` and then reading the contents of the buffer for the argument's value.

3.4 DataType Objects

As the name suggests, the `DataType` of a `FunctionArg` indicates the type of the corresponding CTL function parameter. Class `DataType` is the base of a C++ class hierarchy that describes CTL types. Each of the six fundamental CTL types is represented by a class that is derived from `DataType`:

C++ class	CTL type
`BoolType`	`bool`
`IntType`	`int`
`UIntType`	`unsigned int`
`HalfType`	`half`
`FloatType`	`float`
`VoidType`	`void`

Class `DataTypePtr` has a `cast()` member function template. The application program can call `cast()` to determine whether a `DataTypePtr` points to a `BoolType`, `IntType`, etc. object. Knowing the CTL type of a function argument, the application can cast the buffer for the argument's value to the appropriate C++ type. For example:

```
FunctionArgPtr arg = call->inputArg (0);
DataTypePtr type = arg->type();

if (BoolTypePtr boolType = type.cast<BoolType>())
{
    // CTL type bool

    bool *value = (bool *)arg->data();

    if (arg->isVarying())
    {
        // varying bool

        for (int i = 0; i < n; ++i)
        value[i] = ...;
    }
    else
    {
        // uniform bool

        value[0] = ...;
    }
}
else if (IntTypePtr intType = type.cast<IntType>())
{
    // CTL type int

    ...
```

```
}
...
```

3.5 ArrayType and StructType Objects

C++ classes `ArrayType` and `StructType`, which are derived from class `DataType`, represent CTL arrays and structs. Class `ArrayType` has a fairly large number of member functions, but only a few are relevant here. The rest are used internally by the CTL interpreter:

`size()`	Returns the number of elements in the array.
`elementSize()`	Returns the size, in bytes, of an individual array element.
`objectSize()`	Returns the size, in bytes, of the entire array.
`elementType()`	Returns a `DataTypePtr` that points to the type of the array's elements.

Multi-dimensional CTL array types are represented as nested one-dimensional arrays. For example, CTL type `float[3]` is represented in C++ as an ArrayType object whose `size()`, `elementSize()` and `objectSize()` member functions return 3, 4 and 12 respectively (assuming that the size of a `float` is four bytes). The `elementType()` function returns a pointer to a `FloatType`. CTL type `float[5][3]` is represented by an `ArrayType` whose `size()`, `elementSize()` and `objectSize()` functions return 5, 12 and 60. The `elementType()` function returns a pointer to the `ArrayType` for CTL type `float[3]`.

Accessing an array element requires some address arithmetic in order to find the memory location of the element. The following example sets the value of element `[x][y]` of a varying two-dimensional array of `float` for pixel number `i`:

```
float v = ...;
FunctionArgPtr arg = call->inputArg (...);
DataTypePtr type = arg->type();
char *data = arg->data();

ArrayTypePtr arrayType;
ArrayTypePtr nestedArrayType;
FloatTypePtr elementType;

if ((arrayType = type.cast<ArrayType>()) &&
    (nestedArrayType = arrayType->elementType().cast<ArrayType>()) &&
    (elementType = nestedArrayType->elementType().cast<FloatType>()) &&
    arg->isVarying())
{
    // If we arrive here, we know that arg refers to a
    // varying two-dimensional array of float.
    // Compute the address of array element [x][y] for pixel i.

    char *addr = data +
       i * arrayType->objectSize() +
       x * arrayType->elementSize() +
       y * nestedArrayType->elementSize();

    // Set the array element equal to v.

    *(float *)addr = v;
}
```

C++ class `StructType` describes CTL structs. The following member functions are of interest to application programs that want to pass structs to CTL functions:

`name()`	Returns the name of the struct type.
`objectSize()`	Returns the size, in bytes, of the entire struct.

`members()` Returns a pointer to an `std::vector<Member>` that describes the CTL struct's members. C++ class `Member` has three fields: `name` is an `std::string` that indicates the member's name, `type` is a `DataTypePtr` that describes the member's type, and `offset` indicates the offset, in bytes, of the member from the beginning of the struct.

For example, CTL type

```
struct S
{
    int x;
    int y;
};
```

is represented by a `StructType` object whose `name()` and `objectSize()` member functions return "S", and 8 respectively (assuming that the size of an `int` is four bytes). `members()` returns a two-element vector. The `type` pointers in both elements point to an `IntType`. The `name` fields are set to "x" and "y", and the `offset`s are 0 and 4.

3.6 Error Handling

When an operation such as loading a module or creating a `FunctionCall` object fails, the CTL interpreter reports the failure by throwing a C++ exception. All exceptions thrown by the interpreter are derived from class `std::exception`. Calling an exception's `what()` member function returns an error message that describes problem, for example, "Cannot load CTL module 'oid'. Opening file '/ctl/modules/oid.ctl' failed (permission denied)."

While attempting to load a module, the interpreter may find that the code in the module contains errors, for example, references to undefined variables. In this case the interpreter prints one or more diagnostic messages before throwing an exception. By default, those messages are printed by sending them to the standard error file, `std::cerr`. An application program can redirect the messages by supplying its own message output function:

```
void
myMessageOutput (const std::string &message);
{
    ... // output the message
}
...

Ctl::setMessageOutputFunction (myMessageOutput);
...
```

The mechanism for diagnostic message output is also used by CTL's print statement. Rerouting diagnostic messages from the interpreter also reroutes messages that are printed by the CTL program.

3.7 Breaking Infinite Loops

CTL programs may contain loops, and those loops could be infinite. If a CTL program enters an infinite loop, then the calling C++ application program will hang. In order to prevent this, class `Interpreter` has two member functions that allow the application to abort running CTL programs:

The `setMaxInstCount()` function limits the number of instructions that a CTL program can execute. If a program exceeds the instruction limit, it aborts, and the C++ call to `callFunction()` that started the CTL program throws a `Ctl::MaxInstExc` exception. What exactly an "instruction" is, depends on the interpreter back end. Typically an instruction is a simple operation such as an addition, fetching the value of a variable, or storing a value in a variable. A CTL statement typically corresponds to multiple instructions.

The `abortAllPrograms()` function aborts all currently running CTL programs. The C++ calls to `callFunction()` that started the aborted CTL programs throw `Ctl::AbortExc` exceptions. `abortAllPrograms()` is useful only in multi-threaded C++ programs. Since a thread that is already hanging in `callFunction()` cannot call any other functions, another thread must call `abortAllPrograms()`. (It is not safe to call `abortAllPrograms()` from a signal handler; the call could cause the interpreter to deadlock.)

3.8 Thread-Safety and Parallelism

The CTL interpreter was designed to be thread-safe. The threads in a multi-threaded C++ application program can share a single Interpreter object. The Interpreter object uses mutual exclusion to protect its internal data structures when multiple threads simultaneously call member functions such as `loadModule()` or `newFunctionCall()`.

Multiple application threads can concurrently call different CTL functions or even the same function. In order to do this, each thread must create its own private `FunctionCall` objects. `FunctionCall` objects are not thread-safe, and must not be shared between threads.

With the SIMD interpreter back end, overlapping CTL function calls that are initiated by multiple application threads are executed simultaneously. On a multi-processor computer, this allows the application program to accelerate CTL color transforms by multi-threading: the input image is split into multiple pieces, and each thread applies the transforms to one piece.

Other interpreter back ends may exhibit less parallelism. For example, a GPU-accelerated back end might not be able to run two independent CTL functions simultaneously. In this case overlapping calls initiated by multiple application threads would have to be executed one after the other.

3.9 Accelerating the Transforms

The interpreter's SIMD back end is probably not fast enough for real-time processing of moving images or for high-resolution still images. Writing a significantly faster back end is a non-trivial exercise.

On a computer with multiple processors, transforms can be accelerated by multi-threading. With n processors the image can be split into n tiles, and each tile can be processed by a separate thread. The threads can share a single `Interpreter` object, but each thread must use its own `FunctionCall` objects. Applying the transform to the image should be close to n times as fast as with a single thread.

Depending on how many transforms must be applied to an image, and depending on the complexity of the CTL code for those transforms, multi-threaded execution may still not be fast enough.

Because CTL does not allow static variables, all CTL programs are pure functions: the output of a CTL function call depends only on the CTL source code and on the arguments passed to the function. The output does not depend on side effects of other function calls.

Because transforms are pure functions, any transform or series of concatenated transforms can be tabulated. Typically, the input and output images of a series of transforms each have three channels. The transforms may have other parameters, but their values are uniform and do not vary from pixel to pixel. For a given set of uniform parameter settings, the series of transforms becomes a function that maps a 3D point to another 3D point. Usually this function is continuous and can be approximated by a 3D lookup table. The table can be generated automatically by applying the transforms to an appropriate set of pixels.

Applying the table to an input image takes a constant amount of time, independent of the number or complexity of the transforms from which the table was generated. On systems with GPU support, where the lookup table can be converted into a 3D texture, applying the table to an image can be made extremely fast. Multi-threading and tabulating the transforms should make it possible to use the interpreter with the SIMD back end in real-time applications.

4 Design Decisions

4.1 Limited Set of Language Features

CTL is a fairly simple programming language, and it is missing features that are found in other languages. CTL was designed to do two things: describe color transforms in an unambiguous way, and provide a portable implementation to run those transforms.

Typical color transforms are not algorithmically complex; except for lookup tables, most CTL programs are fairly short. CTL is expressive enough to implement typical color transforms with reasonable effort. A number of features that one would want to include in a general-purpose language were left out because those features would rarely be used by color transforms, and the expected payoff would not justify the implementation effort. Examples include dynamic memory allocation, user-defined function and operator overloading, or classes with member functions and inheritance.

CTL supports constant static data, but it disallows static variables. Without static variables, CTL functions cannot communicate with each other, except by calling one another, and functions cannot retain state between invocations. This allows different interpreter back ends to process pixels in whatever order works best, without affecting the results.

A second reason for disallowing static variables is security: Without static variables, it is very difficult to send or receive data via covert channels such as steganographic encoding in an image's pixels.

4.2 No Support for Execute-Only Color Transforms

In files, CTL programs are always stored as source code. The CTL interpreter does not support an "unreadable" execute-only representation for CTL programs. Such a representation might be desirable in situations where the author of a color transform wants to allow someone else to use the transform without revealing how the transform actually works.

Designing an execute-only representation for CTL programs would not be too hard, but making it resistant to reverse-engineering is nearly impossible. Since color transforms are pure functions, they can be probed and tabulated (see section 3.9). With access to the interpreter's source code one could also write a reverse compiler that could turn an execute-only CTL program into source code that is functionally equivalent to the original.

4.3 Portability

CTL programs are meant to be portable. Except for slight numerical differences (caused by rounding or by acceleration techniques such as tabulating), running a given CTL program should produce the same results, independent of the operating system or the C++ application that hosts the interpreter.

In order to avoid platform-dependencies, the CTL interpreter does not support extension modules written in other languages. CTL programs can only import modules that are also written in CTL. The interpreter has no public interface for calling C++ functions from CTL. (CTL has a standard library of built-in functions, which are, at least for the SIMD back end, written in C++, but this library is compiled into the interpreter, and the CTL-to-C++ function call mechanism is not part of the interpreter's external interface.)

As mentioned above, CTL programs are always stored as source code. This representation is inherently portable. It would be possible to support a portable binary file format for pre-compiled CTL programs. Loading CTL programs from binary files might be faster than directly loading the source code, but since typical CTL programs are fairly short, binary files wouldn't save a lot of time.

4.4 SIMD Back End

At present, the CTL interpreter comes only with one back end, which translates the CTL source into code for a SIMD virtual machine. The virtual machine does not rely on special hardware such as graphics co-processors, or a main processor with SIMD machine instructions. The SIMD interpreter back end is fast enough to be usable, but other implementations could be significantly faster.

The SIMD back end provides a portable reference implementation of CTL. The SIMD back end runs on any platform that supports C++, and it defines a standard for "correct" execution of CTL programs that other back ends must match. By definition, the SIMD reference implementation is always correct.

4.5 Security

Image files and their associated color transforms are meant to be routinely exchanged between various parties. CTL color transforms are programs, and in general, running programs of unknown provenance might carry a risk of running malicious software such as Trojan horses or viruses.

Introducing malicious software into a computer system via CTL programs is very difficult because the language is restricted. CTL programs have no access to the host computer's file system, system calls, clock, communication protocols, or raw memory; and CTL programs cannot load extension modules that are written in languages other than CTL. Barring bugs in the interpreter, the only data that CTL programs can see or modify are the arguments passed to CTL functions that are called directly by a C++ application program. CTL's lack of static variables prevents CTL programs from remembering any data between calls.

CTL programs cannot capture information about the host system, they cannot send data somewhere else, and they cannot gain control of the host machine.

5 CTL Language Reference

This section provides a reference for the CTL programming language. CTL is derived from the C programming language, with some elements borrowed from C++. This reference assumes the reader is familiar with C, and at least syntactical elements of C++. C is defined by standard ISO/IEC 9899, and C++ is defined by standard ISO/IEC 14882.

5.1 Syntactic Notation

The language grammar is included in its entirety in Section 5.8, and is quoted in many of the sections that follow. Syntactic categories are printed in *this font*. Text that appears literally in programs is printed in `this font`. For syntactic rules, alternatives appear on separate lines. The special symbol Ø means that the empty string or "nothing" is a valid alternative. For example:

importList:
    Ø
    importStatement importList

importStatement:
    `import` *stringLiteral* **`;`**

Means that an *importList* can be either the empty string or an *importStatement* followed by an *importList*. An *importStatement* consists of the keyword `import`, followed by a *stringLiteral* and a semicolon.

5.2 Lexical Elements

The text in a CTL files is tokenized using white space to delineate tokens. White space consists of spaces, tabs and new-lines, and tokens are one of the following:

token:
    keyword
    name
    punctuator
    literal

There is only support for ASCII characters; Unicode or other larger character sets are not supported; trigraph sequences, alternate names and universal character names of C++ are not supported.

No C-style preprocessing occurs in CTL, and there are no preprocessing tokens.

5.2.1 Keywords

The CTL keywords are primarily a subset of C++ keywords, with a few additions. Here are keywords in common with C++:

```
bool
const
false
float
for
if
int
long
return
short
signed
struct
true
unsigned
void
```

```
while
namespace
```

These are also CTL keywords:

```
ctlversion
half
input
output
print
uniform
varying
```

The following are reserved as keywords but are not used:

```
break
continue
string
```

Names must begin with a non-digit character A through Z or '_', and in other ways conform to the C specification of *identifiers*, except that there are no predefined identifiers. For example, these are valid names:

```
foo2  foo_bAr_34  __func__  ____
```

and these are not valid names:

```
2foo \foo -fred-
```

5.2.2 Punctuators

Punctuators are the following subset of C:

```
[  ]  (  )  {  }  .  *  +  -  ~  !  /  %  <<  >>  <  >  <=  >=  ==  !=
^  &  |  &&  ||  ;  =  ,
```

The following C punctuators are not part of the CTL language:

```
->  ++  --  ?  :  ...  *=  /=  %=  +=  -=  &=  ^=  |=  <<=  >>=  #  ##
<:  :>  <%  %>  %:  %:%:
```

5.2.2.1 Comments

Both comment types, `//` and `/*`, are supported in CTL as specified in C++.

5.2.3 Literals

There are six types of literals; the five that exist in C++, plus a *half-literal*:

literal:
- *integer-literal*
- *floating-literal*
- *string-literal*
- *boolean-literal*
- *half-literal*

5.2.3.1 Integer Literals

Integer literals include decimal, hexadecimal and octal literals, as specified in C++.

5.2.3.2 Floating Literals

Floating literals behave as specified in C++, except that the F and L suffix (the *floating-suffix*) are not allowed. For example, the following are valid floating literals:

```
1.2  1.2e2  .02  0.02
```

These are not valid floating-literals:

```
1.2F  1.2f 1.2l 1.2L
```

5.2.3.3 Half Literals

Half literals are similar to floating literals, but have a letter h (or H) suffix to differentiate them. Half literals have type `half`. The following are valid half literals:

```
1.2h  1.2e3h  3e-02H
```

5.2.3.4 String Literals

String-literals are as specified in C++, except that there is no support for wide literals; The optional prefix letter L is not supported. CTL is designed for implementing data transforms, so for simplicity string literals can only be used as inputs to the print statement (primarily intended for debugging) and the import directive.

In CTL the character strings used in the import directive are simply string literals. (In C++ there are slightly different rules governing the filename specifier used in the `#include` directive.) The import directive is described in Section 5.3.3.

5.2.3.5 Boolean Literals

Boolean literals, as in C++, consist of `true` and `false` and have type `bool`.

5.3 Basic Concepts

5.3.1 Varying Values

CTL is designed to succinctly describe color transforms. A CTL program describes the operations that are performed on a single pixel of an image. Syntactically, the operations that constitute a color transform take the form of a CTL function. When a transform is applied to an image, the corresponding CTL function is called repeatedly, once for every pixel in the image. The values of some of the function's parameters vary from one pixel to the next, for example, the red, green and blue channels of pixels in an RGB image. These are called “varying” values. The values of other parameters are the same for all pixels; these are called “uniform” values.

Whether the parameters of a CTL function are meant to be varying or uniform matters to a C++ application that calls the CTL function. The application needs to know if a CTL function's parameter refers to per-pixel data (that is, to an image channel) or not. However, the CTL function itself does not depend on, and cannot even determine, whether a value is varying or uniform.

5.3.2 What is A CTL Program

A CTL program at the highest level is a list of function definitions, constant definitions and struct type definitions. Import statements and name spaces are also available to ease modularization. A CTL program may be split into multiple modules.

Note that a CTL program cannot contain global variables. Global constants are allowed, but all variables are local to functions.

5.3.3 Modules

The contents of single CTL source file are referred to as a “module”. Each module has a name. In order to make the constants, struct type definitions and functions in a module available, a CTL interpreter must “load” the module. After a module has been loaded, the functions defined in the module can be called, and the type definitions and constants in the module may be used by other modules.

Modules can load other modules via import directives, which specify the names of the modules to be loaded. Import directives must occur before any of the module's definitions but after an optional CTL version statement:

module:
    ctlVersionStatement importList moduleBody

importList:
    Ø
    importStatement importList

importStatement:
    `import` *stringLiteral* `;`

A module is loaded only once. If subsequent import statements referring to the same module are encountered, the interpreter ignores them.

The mechanism that loads CTL modules is implementation specific; the reference CTL interpreter loads modules from files. The name of the file that contains a given module is formed by appending `".ctl"` to the module name. An environment variable, CTL_MODULE_PATH, specifies a list of directories, or folders, which are searched in order to locate the file (See section 3.2).

5.3.4 CTL Version

An optional CTL version statement at the beginning of a module specifies the version of CTL for which the module was written:

ctlVersionStatement:
    Ø
    `ctlversion` *intLiteral* `;`

The definition of CTL as described in this document has been assigned version number 1. CTL may be extended in the future; when that happens, a new, higher version number will be assigned to the extended definition. Modules written for an extended version of CTL may not run with an interpreter that implements an older version of the language.

If the CTL interpreter encounters a version statement while attempting to load a module, it compares the interpreter's CTL version and the version specified in the version statement. If the specified version is greater than the interpreter's version, the interpreter prints a message to warn the user that loading the module may fail. The interpreter then continues the loading process, regardless of the specified version.

5.3.5 Name Spaces

Only one name space can be defined per module. If there is a name space declaration, it occurs at the beginning of the module and encloses all definitions in the module in curly braces. If no name space is declared in a module, then the module's definitions create names in the global name space.

moduleBody:
    funcConstStructList
    `namespace` *name* `{` *funcConstStructList* `}`

If a name space is specified, every definition in the module belongs to that name space. A particular name space can be used by multiple modules.

Outside of a name space, constants, structures and functions belonging to the name space must be qualified by the name space's name. As in C++ the `::` operator indicates name space qualification:

scopedName:
    name
    name `::` *name*
    `::` *name*

Note that compound statements (also called blocks) such as function bodies create nested anonymous name spaces. Variables defined in functions cannot be accessed outside the function body, but can be accessed without qualifiers by blocks nested in the function.

If an unqualified name (without the `::` operator) is used to refer to a type or an object, the interpreter checks the containing name spaces, starting with the local (most nested) name space and proceeding through less nested name spaces until the global name space. If a name is preceded by a `::` operator, then the name refers to a type or an object in the global namespace. If a name is preceded by a name space name and a `::` operator, then it refers to a type or an object in the corresponding namespace.

For example:

In module `MyModule`:

```
namespace MyLib
{

    const int i1 = 0;
    const int i2 = 0;

    void
    myFunc()
    {
        const int i1 = 1;

        {
            int i = i1;           // assigns 1 to i
            int j = MyLib::i1;    // assigns 0 to j
            int k = i2;           // assigns 0 to k
        }
    }
}
```

In a separate module:

```
import "MyModule";

const int i1 = 3;

void
myFunc()
{
    int i = ::i1;           // assigns 3 to i
    int j = MyLib::i1;      // assigns 0 to j
}
```

There is no equivalent to the `using` declaration from C++.

The `::` operator is only used to reference name space names; CTL does not support classes, and structs support only local data variables.

5.3.6 Definitions

Objects and struct types in CTL are declared and defined at the same time. There are no “forward declarations” that indicate the existence of an object or type, without also defining it.

There can only be one definition of any variable, function or struct type in a CTL program.

The body of a CTL module contains a list of functions, constant definitions and struct type definitions.

funcConstStructList:
    Ø
    funcConstOrStruct funcConstStructList

funcConstOrStruct:
    function
    constantDefinition
    structDefinition

Storage of variables is automatic; storage is guaranteed to be allocated while the variable is in scope and will be deallocated after the variable goes out of scope.

There are no memory allocation or deallocation functions (CTL does not support dynamic storage) and there are no custom constructors or destructors.

Uninitialized data may be set to zero or not, depending on the implementation.

5.3.7 Scope

As in C++, variable scope extends from the point of definition to the end of the containing block, except when a more local variable is present to take precedence. The following function runs to completion (as in C, `assert(x)` does nothing if `x` is true, and aborts the program if `x` is false):

```
void
testScope2()
{
    int i = 0;
    {
        assert( i == 0 );
        int i = 3;
        assert( i == 3 );
    }
    assert(i == 0);
}
```

Functions cannot be nested and thus their scope extends from the point of definition to the end of the module.

5.3.8 Initialization of Constants, Evaluation of Constant Expressions

The initialization of global constants, including the evaluation of expressions required for their initialization, is performed once, when the module that contains the constants is loaded. Expressions that have constant value may be evaluated any time after they are parsed. The reference interpreter evaluates and simplifies expressions with constant value immediately after they are parsed.

5.3.9 Types

CTL is a strongly typed language, with a set of types similar to C++.

5.3.9.1 Fundamental Types

There are six fundamental types in CTL:

```
bool
int
unsigned
unsigned int
half
float
void
```

The `unsigned` type qualifier by itself is shorthand for `unsigned int`. The `int` and `unsigned int` types are 32-bit signed and unsigned integral numbers. The `float` type is a 32-bit floating-point number.

The `half` type is a 16-bit floating point number, which can be used in arithmetic expressions anywhere a `float` can be used. Conversions from `half` to `float` are lossless, but conversions from `float` to `half` may be rounded to the nearest representable `half`. An overflow may occur when converting a `float` to a `half`, if the `float` value is beyond the range of representable `half` values, which will result in the value being an infinity with the same sign as the `float` value.

Type `void` is used only to indicate that a function does not return a value.

String literals are of type “string”, but can only be used in import directives and the print statement. It is not possible to declare variables or function parameters of type string, or structs with members of type string or arrays of strings. There are no operators whose operands are strings.

There is no equivalent to the C++ types `char`, `short` or `long`.

5.3.9.2 Compound Types

There are two kinds of compound types in CTL, structs and arrays. There is no equivalent to C++ pointers, or `enum` and `union` types.

5.3.9.2.1 Struct Types

A struct is a collection of named objects, similar to the C struct. Struct type equivalence in CTL is based on the scoped name of the type, rather than the members of the struct. Two types are the same if they were declared using the same type name in the same name space. Two structs with different scoped type names are not equivalent, even if they have the same members.

5.3.9.2.2 Array Types

An array is a sequence of objects of the same type, as in C. Array types consist of a base type and a sequence of sizes, one size for each dimension of the array.

5.3.10 Const Qualifier

A variable can be defined as `const`, which specifies that it cannot be modified – it cannot be used on the left hand side of an assignment. A variable that has been defined as `const` is called a constant. Function arguments with the `input` qualifier are `const` as well, as explained in section 5.4.3.

5.3.11 Type Conversion

There is no explicit casting operator in CTL. Implicit type conversion occurs in arithmetic expressions, assignments, initialization and function calls.

For the purpose of implicit type conversion, all arithmetic types have a rank. The list of types in increasing rank order is `bool`, `int`, `half`, and `float`. The `unsigned` and `signed int` types have the same rank. As in C++, a type conversion to a higher rank, called a promotion, does not change the value. Other type conversions do occur in some situations, and may change the value represented, as specified in C++.

If a type is used in a place where a type with higher rank is expected, the type is promoted to the expected type. If a type is used in a place where a lower or equal rank is expected, the syntax may or may not be valid depending on the type of expression or statement, as explained in the following sections. If it is valid syntax, a type conversion is performed from the actual type to the expected type. For example:

```
int i = 2.7;
float f = 2.3;

if(f > i)
    i = ~f;  // error int operand expected
```

In the first initializer, the float literal 2.7 is converted to an int type with value 2. In the expression involving the binary `>` operator, `f > i`, the value of `i` has type `int`, and is converted to a float (with no change in value). The bitwise complement operator (`~`) accepts only integer operands, so the expression `~f` is malformed.

It is not possible to convert a compound type to any other type. Any fundamental type can be converted to any other fundamental type except `void`, but some expressions only allow type promotion.

5.4 Definitions

As mentioned previously, the bulk of a CTL module is a list of definitions, including struct type definitions, constant definitions and function definitions. Definitions of variables and struct types also may occur in the body of a function.

5.4.1 Struct Type Definitions

A subset of the syntax for defining structs in C++ is allowed:

structDefinition:
`struct` *name* `{` *structMemberDefinitions* `}` `;`

structMemberDefinitions
Ø
baseType name arraySize `;` *structMemberDefinitions*

For example:

```
struct Mixed
{
   int i;
   float f;
   bool b;
   half ah[2];
};
```

As in C++, this defines a new type, `Mixed`, which can be used in subsequent variable declarations. Definitions for structs can be placed in the global name space, inside named name spaces, or in the anonymous local name spaces of functions or code blocks. The `struct` keyword can only be used to define a new struct type. In contrast to C++, the type definition is always a separate statement from struct variable definitions, and there is no facility for creating unnamed struct types.

5.4.2 Variable Definitions

Variable definitions are specified as follows:

variableDefinition:
baseType name arraySize ;
constness baseType name arraySize = expression `;`
constness baseType name arraySize = compoundInitializer `;`
constness baseType name arraySize `,` *expression* `;`

constness:
Ø
`const`

arraySize:
Ø
`[ ]`
`[` *expression* `]`

If a variable definition occurs outside a function body, the variable type must be `const`. The complete grammar refers to this subset as the *constantDefinition*.

The first three forms of variable definition of fundamental types are syntactically as specified in C++. For example:

```
int i;                // value undefined
half h = 1.201e2;
float f = 2.1*h;
bool b = someBooleanFunc(f,h);
```

Compound objects are initialized using nested lists enclosed in braces. The length and nesting of the initializer must exactly match the length and depth of the compound type being initialized (in contrast to the various initialization options supported in C).

Here are some examples, using the `Mixed` struct defined in the previous section:

```
Mixed m1;
Mixed m2 = {1, 1.0, false, {1,2}};
Mixed m3 = {1, 1.0}; // error in CTL (ok in C++)

Mixed am1[2] =     {{1, 1.0, false, {1,2}},
                    {2, -1.0, true, {-1,-2}}};
Mixed am2[2][1] = {{{1, 1.0, false, {1,2}}},
                   {{2, -1.0, true, {-1,-2}}}};
Mixed am3[1][2] = {{{1, 1.0, false, {1,2}},
                    {2, -1.0, true, {-1,-2}}}};
```

When an initialization is included in the definition, the size specification (the expression between brackets) may be omitted:

```
Mixed am3b[][] = {{{1, 1.0, false, {1,2}},
                   {2, -1.0, true, {-1,-2}}}};
```

An comma-separated initialization expression may also be used to initialize a variable or constant. This allows a constant to be initialized by passing it as an output parameter to a function. For example:

```
void
initArray (output float x[])
{
    for (int i = 0; i < x.size; i = i+1)
        x[i] = i;
}

const float f[100], initArray (f);
```

For the duration of the initialization expression, the variable or constant has non-const type and is defined as though no initializer were present. In the example above, the variable `f` is set by the function `initArray()`. Once the expression is evaluated, `f` has `const` type and may no longer be modified.

5.4.3 Function Definitions

They syntax of function definitions is as follows:

function:
returnType name (*parameterList*) *compoundStatement*

parameterList:
Ø
nonEmptyParameterList

nonEmptyParameterList:
parameter
nonEmptyParameterList , *parameter*

parameter:
`input` *inputParameter*
inputParameter
`output` *outputParameter*

compoundStatement:
{ *statementList* }

Function arguments in CTL are passed by reference. All arguments types have one of two qualifiers, `input` or `output`. Input parameters have const type and cannot be modified in the body of the function. Parameters without qualifiers are input

parameters. Output parameters are intended to store the result of the function, but an output parameter does have an initial value that is determined by the function's caller. Here is an example:

```
bool
foo (output float f, half h)
{
    //  h = 2.0;  // error, h is an input parameter

    if (h > f)
    {
        return false;
    }
    else
    {
        f = h;
        return true;
    }
}

bool
bar ()
{
    float f = 2;
    foo (f, 3);  // returns false
    foo (f, 1);  // assigns 1 to f, returns true
}
```

5.4.3.1 Function Parameters

The syntax of function parameters is as follows:

inputParameter:
varyingHint baseType name arraySize
varyingHint baseType name arraySize = compoundInitializer
varyingHint baseType name arraySize = expression

outputParameter:
varyingHint baseType name arraySize

varyingHint:
Ø
`varying`
`uniform`

The definition of a parameter may include a hint as to whether the parameter is meant to be varying or uniform. If neither `varying` nor `uniform` is specified, then the parameter is assumed to be uniform. The varying or uniform hint is only of interest to C++ applications that want to call CTL functions. The function can actually be called with any argument being varying or uniform, regardless of the hint.

A default value for an input parameter can be specified by providing an initializer or expression assignment (as in C++). If the function is called with fewer arguments than the function has parameters, then default values are used for the unspecified parameters. The expression specifying a parameter's default value must have constant type – it must be possible to evaluate it at compile time. Parameter default expressions cannot refer to other parameters, and output parameters cannot have default values.

```
void
copyWDefaults(output int aiOut[2], int aiIn[2] = {1, 2})
{
    for(int i = 0; i < 2; i=i+1)
    {
        aiOut[i] = aiIn[i];
    }
}
...
```

```
int ai[2];
copyWDefaults(ai, ai2);  // sets ai[0] to 1 and ai[1] to 2
```

5.4.3.2 Variable-Size Arrays

If a function argument is an array type, one or more of its sizes can be unspecified, by leaving the size blank. These array arguments are called variable-size arrays, and allow a function to be called with parameters that have any size in the unspecified dimensions. For example, the previous example could have been written using variable-size arrays and the `size` operator:

```
void
copyVarying(output int aiOut[], int aiIn[] = {1, 2})
{
    if( aiOut.size != aiIn.size)
        return;

    for(int i = 0; i < aiOut.size; i=i+1)
    {
        aiOut[i] = aiIn[i];
    }
}
```

As stated in Section 5.3.9.2.2 the type of an array consists of its base type and a sequence of array sizes, one per dimension. When a function call is parsed, if a particular array dimension is variable-sized, that dimension is not considered in the type checking of the arguments. As long as the base type and dimensions with specified sizes match the passed arguments, the array is considered a type match. For example, `float[][2][]` is considered the same type as `float[1][2][3]` and the same type as `float[100][2][300]`. The above function can be called as follows:

```
    int ai[2];
    int ai2[] = {3,4};
    copyVarying(ai, ai2);
    assert(ai[0] == 3 && ai[1] == 4);

    int ai3[] = {-1};
    int ai4[] = {-2};
    copyVarying(ai3);      // returns without copying
    assert(ai3[0] == -1);  // because aiIn[] uses default
    copyVarying(ai3, ai4);
    assert(ai3[0] == -2);
```

In the body of a function definition, variable array types do not match any other type – not even other variable-size arrays. As a result, expressions with varying types cannot be used in some statements, such as assignment or initialization.

```
void
varyingBody(output int ai1[][2],
            output int ai2[][3],
            output int ai3[][2])
{
    ai1 = ai2;             // error - int[][2] != int[][3]
    ai1 = ai3;             // error - int[][2] != int[][2]
    int aiL1[][2] = ai1;   // error

    ai1[0] = ai3[0];       // ok - int[2] == int[2]

    int aiL2[ai1.size][2]; // error - size unknown at
                           // compile time
}
```

Only function arguments can have variable sizes; it is not possible to define a variable-size local variable.

The `size` operator returns the size of the left-most dimension for a given-type. To extract the size of other dimensions, the size operator must be used in combination with the array index operator. If the dimension is variable, the `size` operator is evaluated at run time. The `size` operator evaluates to an integer at compile time if the size of the tested dimension is not variable.

```
void
testArraySize(int aiArg[1][][2])
{
    assert(aiArg.size == 1);       // = 1 at compile time
    assert(aiArg[0].size == 3);    // unknown at compile time
    assert(aiArg[0][0].size == 2); // = 2 at compile time
}
...

int ai3d [1][3][2];
testArraySize(ai3d);

int ai3d2 [1][2][2];
testArraySize(ai3d2); // assert fails
```

Variable-sized arrays cannot be returned by functions.

5.5 Statements

The body of a function contains a list of statements. Statements include the variable and struct type definitions described above, plus familiar C-style statements for other purposes, such as control flow, assignment, and expression:

statement:
variableDefinition
structDefinition
compoundStatement

whileStatement
forStatement
ifStatement

assignment
expressionStatement

nullStatement
printStatement
returnStatement

There are no `switch`, `do`, `jump`, `break`, or `continue` statements.

5.5.1 Compound Statements

Compound statements, also called code blocks, are curly braces containing a list of statements. As in C/C++, they are used in the bodies of other control structures, such as for loops and if statements, but may also stand on their own.

compoundStatement:
{ *statementList* }

statementList:
Ø
statement statementList

Statements contained in the *statementList* of a compound statement have their own local name space, as described in Section 5.3.5.

5.5.2 While Statements

The `while` statement has the same form as in C++:

whileStatement:

`while` (*expression*) *statement*

The while statement is evaluated over and over until the expression is `false`. During each evaluation, the expression is evaluated and its type is converted to type `bool`. If the result is `true`, the trailing statement is evaluated.

5.5.3 For Statements

The `for` statements have the following form:

forStatement:
`for` (*forInitStatement* `;` *expression* `;` *forUpdateStatement*) *statement*

forInitStatement:
variableDefinition
assignment
expressionStatement

forUpdateStatement:
simpleAssignment
simpleExpressionStatement

A `for` statement is similar to its namesake in C/C++, in that it contains an initialization, conditional test and an update statement, but the form is more restricted. The initialization can contain one variable definition, assignment or expression (as opposed to lists of them). The update statement consists of a single assignment or expression statement. Expressions and expression statements are explained below.

5.5.4 If Statements

The `if` statements have the form as specified in C++.

ifStatement:
`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

The expression is evaluated and its type converted to a Boolean. If the expression is `true`, the statement is executed. If the expression is `false`, the optional statement following the `else` is executed.

5.5.5 Assignments

The assignment statement sets the value of an object on the left hand side.

assignment:
simpleAssignment `;`

simpleAssigment:
expression = *expression*

The left hand side must evaluate to an object that is modifiable. The right hand side must be either the same type as the left hand side or a type that can be converted to the left hand side type. If the right hand side is not the same type as the left hand side, a type conversion will occur.

Assignments can use any modifiable type on the left hand side, including compound types. For example:

```
void
sampleAssignments()
{
    int ai1[1][2];
    int ai2[2] = {1,2};
    ai1[0] = ai2;
}
```

Notice also the example in Section 5.6.1.

There are no compound assignment operators, such as += or −=.

In CTL, assignments are statements, not expressions; a statement such as

```
a = b = c;
```

is not allowed.

5.5.6 Expression Statements

As in C++ any expression can stand alone as a statement with a semicolon appended.

expressionStatement:
    simpleExpressionStatement ;

simpleExpressionStatement:
    expression

Expressions are discussed in Section 5.6. The expression statement is useful only for its side effects, for example in calling a function with output arguments.

5.5.7 Null Statements

As in C or C++, a blank statement followed by a semicolon can be used wherever a statement may occur, to indicate that no operation is to be performed.

nullStatement:
    ;

5.5.8 Print Statements

A print statement is provided to assist with debugging CTL programs, or to provide very limited user feedback. A print statement outputs the result of expressions to the standard output:

printStatement:
    print (*exprList*) ;

exprList:
    Ø
    nonEmptyExprList

nonEmptyExprList:
    expression
    nonEmptyExprList , *expression*

Each expression in the *exprList* can have any of the fundamental types or be a string literal. The underlying implementation should convert each of the fundamental types to characters in some intuitive way. The reference implementation calls the default C++ stringstream << operator. For example:

```
bool b = false;
float f = -.00000012;
print("b = ", b, ", f = ", f);
print(", random literals: ", 4, ", ", 3.00977h,"\n");
```

causes the following to be generated in the reference implementation:

```
b = 0, f = -1.2e-07, random literals: 4, 3.00977
```

The print statement does not print compound types.

If a varying expression is passed to the print statement, the print statement may print multiple values, one per pixel. Print statements do not in any way affect the values returned by CTL transformations, so the exact formatting is not specified. This flexibility makes the print statement easier to implement, but limits its usefulness beyond debugging.

The reference implementation is described here as an example; the details in the remainder of this section are not part of the language specification.

The reference implementation prints a varying expression inside of brackets with the word `varying` appearing first. CTL functions are run simultaneously on arrays of values of a particular length, and the function calls are called with arrays of this length until pixels have been processed. The print statement outputs all the values at once from a particular varying function array's function call.

For example, if `f` is a varying argument the implementation may call the function `printExample` on arrays of `f` values of length `5`. A particular evaluation instance with values `1.2`, `0`, `0`, `-1.001`, and `0`, for the following function:

```
void
printExample(input varying float f)
{
    print("all f: ", f, "\n");
    if(f > 1)
        print ("f > 1: ", f, "\n");
}
```

generates the following output:

```
all f: [varying (0, 1.2) (1, 0) (2, 0) (3, -1.001) (4, 0)]
f > 1: [varying (0, 1.2)]
```

In practice the run time engine is most efficient when it runs on thousands of values, and images have millions of pixels. When the print function is called on a varying value in practice, the reference run-time engine generates thousands of separate print statements, each of which is very long – only useful as a focused debugging technique.

5.5.9 Return Statements

The return statement is used to terminate a function and to optionally return a value:

returnStatement:
`return ;`
`return` *expression* `;`

As in C++, the expression can be omitted only in functions with `void` type.

5.6 Expressions

An expression is the combination of values, objects and operands that specify a value or call a function, as specified in the C and C++ standards.

The expression grammar is simpler than the C++ grammar because there are fewer operators and fewer side effects from expressions. The language does not have the `sizeof` operator, conditional operator (`x? a:b`), comma operator or any compound assignment operators (`+=`, `^=` etc.).

There are also no explicit cast operators or function pointers. Side effects from expressions are limited to calls of functions with output parameters and the print statement. There are no assignment expressions – assignments occur in assignment statements only and cannot be nested.

Operator precedence is as specified in C++.

5.6.1 Primary Expressions

A primary expression is the simplest expression building block, consisting of a literal, a variable name, function call or some combination of member names or array access expressions.

primaryExpression:
`true`

`false`
intLiteral
floatLiteral
halfLiteral
stringLiteral
`(` *expression* `)`
scopedName memberArrayExpression
scopedName `(` *exprList* `)`

memberArrayExpression:
Ø
`.` *name memberArrayExpression*
`. size`
`[` *expression* `]` *memberArrayExpression*

Struct members and functions must be referred to explicitly by name. The member array expressions and function call are the only postfix operators supported. As in C++, struct member access and array subscripting can be arbitrarily nested, and multidimensional arrays are indexed by multiple repeated subscripts. The following demonstrates some nested primary expressions used in assignments.

```
void
sampleExpressions()
{
    struct Inside   {int i[2];};
    struct Outside  {Inside ain[2][3];};

    Inside in = {{1,2}};

    Outside out = {{{{{0,1}},{{2,3}},{{4,5}}},
                    {{{6,7}},{{8,9}},{{10,11}}}}};

    in = out.ain[0][1];
    assert(in.i[0] == out.ain[0][1].i[0]);
}
```

As described in Section 5.4.3.2, the size of an array can be determined using the `size` operator. Structs can not have a member named `size`, but variables can be named `size`.

5.6.2 Arithmetic Expressions

The following table summarizes the arithmetic operators that are supported. It is the complete set of arithmetic operators with the same precedence as in C++, but the type conversion process is somewhat different from C++.

Name in Grammar	Operator	Result	Operand Type	Result Type
unaryExpression	`- x`	negative	arithmetic	type of `x`
	`~ x`	bitwise complement	integer	type of `x`
	`! x`	bitwise not	arithmetic	type of `x`
mulitplicativeExpression	`x * y`	multiplication	arithmetic	higher rank type of `x` or `y`
	`x / y`	division	arithmetic	higher rank type of `x` or `y`
	`x % y`	remainder of division of `x` by `y`	integer	higher rank type of `x` or `y`
additiveExpression	`x + y`	addition	arithmetic	higher rank type of `x` or `y`
	`x - y`	subtraction	arithmetic	higher rank type of `x` or `y`
shiftExpression	`x << y`	bitwise shift `x` left `y` digits	integer	higher rank type of `x` or `y`
	`x >> y`	bitwise shift `x` right `y` digits	integer	higher rank type of `x` or `y`

Name in Grammar	Operator	Result	Operand Type	Result Type
relationalExpression	`x < y`	`true` if `x` is less than `y`	arithmetic	higher rank type of `x` or `y`
	`x > y`	`true` if `x` is greater than `y`	arithmetic	higher rank type of `x` or `y`
	`x <= y`	`true` if `x` is less than `y` or `x` equals `y`	arithmetic	higher rank type of `x` or `y`
	`x >= y`	`true` if `x` is greater than `y` or `x` equals `y`	arithmetic	higher rank type of `x` or `y`
equalityExpression	`x == y`	`true` if `x` equals `y`	arithmetic	`bool`
	`x != y`	`true` if `x` is not equal to `y`	arithmetic	`bool`
bitAndExpression	`x & y`	bitwise and	integer	higher rank type of `x` or `y`
bitOrExpression	`x | y`	bitwise or	integer	higher rank type of `x` or `y`
bitXorExpression	`x ^ y`	bitwise exclusive or	integer	higher rank type of `x` or `y`
andExpression	`x && y`	sequential and: `false` if `x` converted to `bool` is `false`, otherwise `y`, converted to `bool`. If `x` is `false`, then `y` is not evaluated.	arithmetic	`bool`
orExpression	`x || y`	sequential or: `true` if `x` converted to `bool` is `true`, otherwise `y`, converted to `bool`. If `x` is `true`, then `y` is not evaluated.	arithmetic	`bool`

In expressions with two operands, if the operands are not the same type, a type conversion will occur. If possible, the left operand is promoted to the type of the right operand. Otherwise, if possible, the right operand is promoted to the type of the left operand. If neither promotion is possible, the expression is not valid.

In expressions requiring integer operands, Boolean operands are promoted to integers. Such operations are not permitted on operands with `float` or `half` types.

The functions in Section 5.7.2 can be used to test if an arithmetic expression generates a result that is a not a number. It is recommended that the underlying platform does not attempt to trap floating point exceptions during execution of a CTL program.

5.7 Standard Library

CTL has a standard library of built-in functions and numeric constants, which are described below.

5.7.1 Numeric Constants

Type	Name	Value
`float`	`M_E`	e (approximately 2.7182818)
`float`	`M_PI`	π (approximately 3.1415927)
`float`	`FLT_MAX`	the largest positive number that is exactly representable as a `float`
`float`	`FLT_MIN`	the smallest positive number that is exactly representable as a normalized `float`
`float`	`FLT_EPSILON`	the smallest positive ε such that $1+\varepsilon$ is exactly representable as a `float`
`float`	`FLT_POS_INF`	$+\infty$
`float`	`FLT_NEG_INF`	$-\infty$
`float`	`FLT_NAN`	a quiet NaN (not-a-number)
`half`	`HALF_MAX`	the largest positive number that is exactly representable as a `half`
`half`	`HALF_MIN`	the smallest positive number that is exactly representable as a `half`
`half`	`HALF_EPSILON`	the smallest positive ε such that $1+\varepsilon$ is exactly representable as a `half`
`half`	`HALF_POS_INF`	$+\infty$
`half`	`HALF_NEG_INF`	$-\infty$
`half`	`HALF_NAN`	a quiet NaN (not-a-number)
`int`	`INT_MAX`	the largest positive integer that can be stored in an `int`
`int`	`INT_MIN`	the largest negative integer that can be stored in an `int`
`unsigned int`	`UINT_MAX`	the largest integer that can be stored in an `unsigned int`

5.7.2 Floating-Point Number Classification

```
bool isfinite_f (float x);
bool isfinite_h (half x);
```
returns `true` if `x` is finite, that is, if `x` is not $+\infty$, $-\infty$ or a NaN; returns `false` otherwise

```
bool isnormal_f (float x);
bool isnormal_h (half x);
```
returns `true` if `x` is a normalized number; returns `false` otherwise

```
bool isnan_f (float x);
bool isnan_h (half x);
```
returns `true` if `x` is a NaN; returns `false` otherwise

```
bool isinf_f (float x);
bool isinf_h (half x);
```
returns `true` if `x` is $+\infty$ or $-\infty$, that is, if `x` is an infinity; returns `false` otherwise

5.7.3 Elementary Functions

`float acos (float x);`
returns the arc cosine of `x`. The result is in radians, and between 0 and π.

`float asin (float x);`
returns the arc sine of `x`. The result is in radians, and between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$.

`float atan (float x);`
returns the arc tangent of `x`. The result is in radians, and between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$.

`float atan2 (float y, float x);`
returns the arc tangent of $\frac{y}{x}$, but takes the signs of `x` and `y` into account to determine the quadrant of the result. The result is in radians, and between $-\pi$ and $+\pi$.

`float cos (float x);`
returns the cosine of `x`, where `x` is given in radians

`float sin (float x);`
returns the sine of `x`, where `x` is given in radians

`float tan (float x);`
returns the tangent of `x`, where `x` is given in radians

`float cosh (float x);`
returns $\frac{e^x + e^{-x}}{2}$

`float sinh (float x);`
returns $\frac{e^x - e^{-x}}{2}$

`float tanh (float x);`
returns $\frac{e^x - e^{-x}}{e^x + e^{-x}}$

`float exp (float x);`
returns e^x

`half exp_h (float x);`
a faster version of `exp(x)` that returns a value of type `half`

`float log (float x);`
returns the natural logarithm of `x`

`float log_h (half x);`
a faster version of `log(x)` where `x` is of type `half`

`float log10 (float x);`
returns the base-10 logarithm of `x`

`float log10_h (half x);`
a faster version of `log10(x)` where `x` is of type `half`

`float pow (float x, float y);`
returns x^y

`half pow_h (half x, float y);`
a faster version of `pow(x,y)` where `x` and the return value are of type `half`

`float pow10 (float x);`
returns `pow(10,x)`

`half pow10_h (float x);`
a faster version of `pow10(x)` that returns a value of type `half`

`float sqrt (float x);`
returns $\sqrt{x}$

`float fabs (float x);`
returns the absolute value of `x`

`float floor (float x);`
returns an integral value, i, such that $0 \le x - i < 1$

`float fmod (float x, float y);`
returns $x - n \cdot y$, where n is an integer such that n has the same sign as $\frac{x}{y}$ and $0 \le \left|\frac{x}{y}\right| - |n| < 1$

`float hypot (float x, float y);`
returns $\sqrt{x^2 + y^2}$

5.7.4 Operations on 3D Vectors and 4×4 Matrices

`float[4][4] mult_f44_f44 (float A[4][4], float B[4][4]);`
matrix-times-matrix multiplication, returns $A \cdot B$

```
float[4][4] mult_f_f44 (float f, float A[4][4]);
```
scalar-times-matrix multiplication, returns $f \cdot B$

```
float[4][4] add_f44_f44 (float A[4][4], float B[4][4]);
```
component-wise matrix addition, returns $A + B$

```
float[4][4] invert_f44 (float A[4][4]);
```
matrix inversion, returns A^{-1} if A is invertible, or I if A is not invertible

```
float[3] mult_f3_f44 (float x[3], float A[4][4]);
```
vector-times-matrix multiplication; returns $x \cdot A$, where x and the result, y, are interpreted as row vectors with homogeneous coordinates, `(x[0], x[1], x[2], 1)` and `(y[0], y[1], y[2], 1)`

```
float[3] mult_f_f3 (float f, float x[3]);
```
scalar-times-vector multiplication, returns $f \cdot x$

```
float[3] add_f3_f3 (float x[3], float y[3]);
```
vector addition, returns $x + y$

```
float[3] sub_f3_f3 (float x[3], float y[3]);
```
vector subtraction, returns $x - y$

```
float[3] cross_f3_f3 (float x[3], float y[3]);
```
cross product, returns $x \times y$

```
float dot_f3_f3 (float x[3], float y[3]);
```
dot product, returns $x \cdot y$

```
float length_f3 (float x[3]);
```
vector length, returns $\sqrt{x \cdot x}$

5.7.5 Lookup Tables and Scattered Data Interpolation

```
float lookup1D (float table[], float pMin, float pMax, float p);
```

Function `lookup1D()` performs a one-dimensional table lookup with linear interpolation.

`lookup1D()` returns `f(p)`, where `f` is a piece-wise linear function. For `pMin` ≤ `p` < `pMax`, `f(p)` is equal to

```
table[i] * (1-s) + table[i+1] * s
```

where

```
t = (p - pMin) / (pMax - pMin) * (table.size-1)
i = floor (t)
s = t - i
```

For `p` < `pMin` and `p` ≥ `pMax`, `f(p)` is equal to `table[0]` and `table[table.size-1]` respectively.

```
float[3] lookup3D_f3
        (float table[][][][3],
         float pMin[3],
         float pMax[3],
         float p[3]);
```

Function `lookup3D_f3()` performs a three-dimensional table lookup with trilinear interpolation:

`lookup3D_f3()` returns `f(p)`, where `f` is a function that maps 3D points to 3D points. The function is defined as follows:

`table`, `pMin` and `pMax` define an axis-aligned 3D grid of `(iMax+1)` by `(jMax+1)` by `(kMax+1)` evenly spaced points, where

```
iMax = table.size - 1
jMax = table[0].size - 1
kMax = table[0][0].size - 1
```

and grid point `(i,j,k)` is at location

```
(pMin[0] + (pMax[0]-pMin[0]) * i/iMax,
 pMin[1] + (pMax[1]-pMin[1]) * j/jMax,
 pMin[2] + (pMax[2]-pMin[2]) * k/kMax)
```

If a point, `p`, is at the same location as grid point `(i,j,k)`, then `f(p)` is equal to `table[i][j][k]`. If `p` is not at a grid point location, then `f(p)` is trilinearly interpolated from the eight nearest grid points:

```
f(p) =
    ((table[i][j][k]     * (1-si) + table[i+1][j][k]     * si) * (1-sj) +
     (table[i][j+1][k]   * (1-si) + table[i+1][j+1][k]   * si) *  sj   ) * (1-sk) +
    ((table[i][j][k+1]   * (1-si) + table[i+1][j][k+1]   * si) * (1-sj) +
     (table[i][j+1][k+1] * (1-si) + table[i+1][j+1][k+1] * si) *  sj   ) *  sk
```

where

```
ti = (p[0] - pMin[0]) / (pMax[0] - pMin[0]) * iMax
i = floor (ti)
si = ti - i

tj = (p[1] - pMin[1]) / (pMax[1] - pMin[1]) * jMax
j = floor (tj)
sj = tj - j

tk = (p[2] - pMin[2]) / (pMax[2] - pMin[2]) * kMax
k = floor (tk)
sk = tk - k
```

If `p` is outside the grid, then `f(p)` is equal to `f(q)`, where point `q` is at position

```
(max (pMin[0], min (pMax[0], p[0])),
 max (pMin[1], min (pMax[1], p[1])),
 max (pMin[2], min (pMax[2], p[2])))
```

```
void lookup3D_f
       (float table[][][][3],
        float pMin[3],
        float pMax[3],
        float p0, float p1, float p2,
        output float q0, output float q1, output float q2);

void lookup3D_h
       (float table[][][][3],
        float pMin[3],
        float pMax[3],
        half p0, half p1, half p2,
        output half q0, output half q1, output half q2);
```

Functions `lookup3D_f()` and `lookup3D_h()` are variants of `lookup3D_f3()` that are more convenient to call when the three components of point `p` happen to reside in three `float` or `half` variables rather than in a three-element array. `lookup3D_f()` and `lookup3D_h()` are both equivalent to the following code:

```
float p[3] = {p0, p1, p2};
float q[3] = lookup3D_f3 (table, pMin, pMax, p);
q0 = q[0];
q1 = q[1];
q2 = q[2];
```

```
void scatteredDataToGrid3D (float data[][2][3],
                            float pMin[3],
                            float pMax[3],
                            output float grid[][][][3]);
```

Function `scatteredDataToGrid3D()` performs three-dimensional scattered data interpolation and builds a table that can be used as an input for the three-dimensional table-lookup functions listed above.

`data` is an array of pairs of 3D points. Each pair, `data[i]`, represents a sample of an unknown function: the value of the function at `data[i][0]` is `data[i][1]`.

`scatteredDataToGrid3D()` approximates this unknown function by interpolating the samples stored in data. First, a smooth function, `f`, is constructed such that `f(data[i][0])` is equal to `data[i][1]` for each `i` with $i \leq 0$ and `i<data.size`. Function `f` is then sampled at regular intervals, and the result is stored in array `grid`:

```
int iMax = grid.size - 1;
int jMax = grid[0].size - 1;
int kMax = grid[0][0].size - 1;

for (i = 0; i <= iMax; i = i+1)
    for (j = 0; j <= jMax; j = j+1)
        for (k = 0; k <= kMax; k = k+1)
        {
            float p[3] =
            {
                pMin[0] + (pMax[0] - pmin[0]) * i/iMax;
                pMin[1] + (pMax[1] - pmin[1]) * j/jMax;
                pMin[2] + (pMax[2] - pmin[2]) * k/kMax;
            };

            grid[i][j][k] = f(p);
        }
    }
}
```

5.7.6 Conversions between Standard Color Spaces

```
    struct Chromaticities
    {
        float red[2];
        float green[2];
        float blue[2];
        float white[2];
    };

float[4][4] RGBtoXYZ (Chromaticities c, float Y);
float[4][4] XYZtoRGB (Chromaticities c, float Y);
```

Functions `RGBtoXYZ()` and `XYZtoRGB()` compute matrices for converting between the CIE 1931 XYZ color space and an RGB color space with arbitrary primaries and an arbitrary white point.

If `c` defines the CIE xy coordinates of the primaries and the white point of the RGB space, and `Y` defines the luminance of the RGB triple (1,1,1), or "white", then `RGBtoXYZ(c,Y)` returns a matrix, M, so that multiplying an RGB value by M produces an equivalent XYZ value.

`XYZtoRGB(c,Y)` returns M^{-1}. Multiplying an XYZ value by M^{-1} produces an equivalent RGB value.

The following example converts the RGB value (0.1, 1.2, 0.4) to CIE XYZ. The primaries and white point of the RGB space match Recommendation ITU-R BT.709-5:

```
const Chromaticities c =
{
    {0.6400, 0.3300},       // red x and y
    {0.3000, 0.6000},       // green x and y
    {0.1500, 0.0600},       // blue x and y
    {0.3127, 0.3290}        // white x and y
};

const float Y = 100.0;      // 100 nits

float M[4][4] = RGBtoXYZ (c, Y);

float RGB[3] = {0.1, 1.2, 0.4};
float XYZ[3] = mult_f3_f44 (RGB, M);
```

Note: The reason why RGBtoXYZ() and XYZtoRGB() return matrices instead of directly converting between RGB and XYZ values is speed. A vector-times-matrix multiplication is faster than computing the matrix. The matrix can be built once, and then re-used many times.

```
float[3] XYZtoLuv (float XYZ[3], float XYZn[3]);
float[3] LuvtoXYZ (float Luv[3], float XYZn[3]);
```

Conversion between CIE XYZ and CIE L*u*v*:

Given an XYZ tristimulus, `XYZ`, and a white stimulus, `XYZn`, `XYZtoLuv(XYZ,XYZn)` returns an L*u*v* triple that is equivalent to `XYZ`.

Given an L*u*v* triple, `Luv`, and a white stimulus, `XYZn`, `LuvtoXYZ(Luv,XYZn)` returns an XYZ tristimulus that is equivalent to `Luv`.

```
float[3] XYZtoLab (float XYZ[3], float XYZn[3]);
float[3] LabtoXYZ (float Lab[3], float XYZn[3]);
```

Conversion between CIE XYZ and CIE L*a*b*:

Given an XYZ tristimulus, `XYZ`, and a white stimulus, `XYZn`, `XYZtoLab(XYZ,XYZn)` returns an L*a*b* triple that is equivalent to `XYZ`.

Given an L*a*b* triple, `Lab`, and a white stimulus, `XYZn`, `LabtoXYZ(Lab,XYZn)` returns an XYZ tristimulus that is equivalent to `Lab`.

5.7.7 Assertion

```
void assert (bool assumption);
```

If `assumption` is `true`, then `assert()` does nothing and returns. If `assumption` is `false`, then `assert()` aborts the calling program. ("Abort" means that the CTL program terminates immediately, and the CTL interpreter throws a C++ exception of type `Iex::LogicExc`.)

The purpose of `assert()` is to make a programmer's assumptions about his or her program explicit, and to detect situations where those assumptions are violated.

Example:

```
float
mySqrt (float x)
{
    assert (x >= 0);

    if (x == 0)
        return 0;

    float a = 1;
    float b = x;

    while (b < 1-FLT_EPSILON || b > 1+FLT_EPSILON)
    {
        a = 0.5 * (a + x/a);
        b = x / (a*a);
    }

    return a;
}
```

If `mySqrt(x)` is called with $x \geq 0$, the `while` loop quickly converges on $\sqrt{x}$, and the function returns. If `x` is less than zero then $\sqrt{x}$ is undefined and the loop does not terminate. The call to `assert()` at the start of `mySqrt()` states explicitly that `x` must not be less than zero; if `mySqrt(x)` is called with `x` less than zero, then the calling program is aborted.

5.8 Complete Grammar

module:
    ctlVersionStatement importList moduleBody

ctlVersionStatement:
    Ø
    `ctlversion` *intLiteral* `;`

importList:
    Ø
    importStatement importList

importStatement:
    `import` *stringLiteral* `;`

moduleBody:
    funcConstStructList
    `namespace` *name* `{` *funcConstStructList* `}`

funcConstStructList:
    Ø
    funcConstOrStruct funcConstStructList

funcConstOrStruct:
    function
    constantDefinition
    structDefinition

function:
    returnType name `(` *parameterList* `)` *compoundStatement*

parameterList:
    Ø
    nonEmptyParameterList

nonEmptyParameterList:
    parameter
    nonEmptyParameterList `,` *parameter*

parameter:
    `input` *inputParameter*
    inputParameter
    `output` *outputParameter*

inputParameter:
    varyingHint baseType name arraySize
    varyingHint baseType name arraySize = *compoundInitializer*
    varyingHint baseType name arraySize = *expression*

outputParameter:
    varyingHint baseType name arraySize

compoundStatement:

{ *statementList* }

statementList:
Ø
statement statementList

statement:
variableDefinition
structDefinition
assignment
expressionStatement
compoundStatement
forStatement
ifStatement
nullStatement
printStatemen
returnStatement
whileStatement

constantDefinition:
`const` *baseType name arraySize = expression* `;`
`const` *baseType name arraySize = compoundInitializer* `;`

variableDefinition:
baseType name arraySize ;
constness baseType name arraySize = expression `;`
constness baseType name arraySize = compoundInitializer `;`
constness baseType name arraySize `,` *expression* `;`

assignment:
simpleAssignment `;`

simpleAssigment:
expression = expression

expressionStatement:
simpleExpressionStatement `;`

simpleExpressionStatement:
expression

forStatement:
`for` `(` *forInitStatement* `;` *expression* `;` *forUpdateStatement* `)` *statement*

forInitStatement:
variableDefinition
assignment
expressionStatement

forUpdateStatement:
simpleAssignment
simpleExpressionStatement

ifStatement:
if (expression) statement
`if` `(` *expression* `)` *statement* `else` *statement*

returnStatement:
`return ;`
`return` *expression* `;`

printStatement:
`print (` *exprList* `) ;`

nullStatement:
`;`

whileStatement:
`while (` *expression* `)` *statement*

expression:
orExpression

orExpression:
andExpression
orExpression `||` *andExpression*

andExpression:
bitOrExpression
andExpression `&&` *bitOrExpression*

bitOrExpression:
bitXorExpression
bitOrExpression `|` *bitXorExpression*

bitXorExpression:
bitAndExpression
bitXorExpression `^` *bitAndExpression*

bitAndExpression:
equalityExpression
bitAndExpression `&` *equalityExpression*

equalityExpression:
relationalExpression
equalityExpression `==` *relationalExpression*
equalityExpression `!=` *relationalExpression*

relationalExpression:
shiftExpression
relationalExpression `<` *shiftExpression*
relationalExpression `>` *shiftExpression*
relationalExpression `<=` *shiftExpression*
relationalExpression `>=` *shiftExpression*

shiftExpression:
additiveExpression
shiftExpression `<<` *additiveExpression*
shiftExpression `>>` *additiveExpression*

additiveExpression:
multiplicativeExpression
additiveExpression `+` *multiplicativeExpression*

additiveExpression `-` *multiplicativeExpression*

multiplicativeExpression:
unaryExpression
multiplicativeExpression `*` *unaryExpression*
multiplicativeExpression `/` *unaryExpression*
multiplicativeExpression `%` *unaryExpression*

unaryExpression:
primaryExpression
`-` *primaryExpression*
`~` *primaryExpression*
`!` *primaryExpression*

primaryExpression:
true
false
intLiteral
floatLiteral
halfLiteral
stringLiteral
`(` *expression* `)`
scopedName memberArrayExpression
scopedName `(` *exprList* `)`

memberArrayExpression:
Ø
`.` *name memberArrayExpression*
`. size`
`[` *expression* `]` *memberArrayExpression*

initializer:
`{` *exprList* `}`

scopedName:
name
name `::` *name*

exprList:
Ø
nonEmptyExprList

nonEmptyExprList:
expression
nonEmptyExprList `,` *expression*

returnType:
varyingHint void
varyingHint baseType arraySize

constness:
Ø
`const`

baseType:
`bool`

`int`
`unsigned`
`unsigned int`
`half`
`float`
baseStructName

compoundInitializer:
`{` *initializerList* `}`
`{` *initializerList* `,` *compoundInitializer* `}`

initializerList:
intializer
initializerList `,` *initializer*

varyingHint:
Ø
`varying`
`uniform`

arraySize:
Ø
`[ ]`
`[` *expression* `]`

structDefinition:
`struct` *name* `{` *structMemberDefinitions* `}` `;`

structMemberDefinitions
Ø
baseType name arraySize `;` *structMemberDefinitions*

Appendix A Simplified API for OpenEXR

The CTL interpreter's C++ interface for passing function call arguments to and from CTL is fairly complex, mostly because the interface must be able to accommodate cases such as varying structures and arrays, as well as limits on the number of pixels that the interpreter back end can handle simultaneously. Given a specific application, it is usually possible to simplify the argument passing mechanism by writing an abstraction layer on top of the basic interface.

The sample source code mentioned in Appendix C includes an application-specific interface for images that are stored in the OpenEXR file format. This OpenEXR interface is not part of the CTL interpreter, and the interpreter does not depend on the OpenEXR file format. OpenEXR is a high-dynamic-range image file format that is frequently employed by the visual effects industry. (For more information on OpenEXR, see http://www.openexr.com/)

A.1 C++ Interface

The OpenEXR interface consists of a single function, called `applyTransforms()`. The function applies a series of CTL functions to the pixels in an OpenEXR frame buffer and places the results in another OpenEXR frame buffer. Two OpenEXR `Header` objects supply uniform input data to the CTL functions. Uniform output data are stored in a third `Header` object.

Function `applyTransforms()` has eight parameters:

`interpreter`	The instance of the CTL interpreter that will execute the color transformation functions.
`transformNames`	A list of CTL function names. The corresponding CTL functions will be called in the order in which their names appear in this list.
`transformWindow`	The region in the input and output frame buffers that contains the pixel data that will be read and written by the CTL functions. Typically this will be the same as the input image's data window.
`envHeader`	An OpenEXR header that contains information about "the environment", for example, display primary chromaticities and white point.
`inHeader`	An OpenEXR header that describes the pixels in the input frame buffer. This will typically be the header of the image file that is the source of the pixels in the input frame buffer.
`inFb`	The input frame buffer; contains the pixels that are to be processed.
`outHeader`	An OpenEXR header where the values of the non-varying output parameters of the CTL functions will be stored.
`outFb`	The output frame buffer; holds the pixels output by the CTL functions.

The `applyTransforms()` function first loads the CTL modules that contain the functions listed in `transformNames`. Each function is assumed to live in a module with the same name as the function. If `transformNames` contains a function `foo()`, then module `foo`, in file `foo.ctl`, is loaded.

`applyTransforms()` then calls each of the CTL functions listed in `transformNames`. The values for the functions' input parameters come from the output arguments of the previous function in the list, from `envHeader`, from `inHeader`, or from `inFb`. The values of some of the functions' output parameters are stored in `outHeader` or `outFb`.

A.2 Matching CTL Parameters, Image Channels and Header Attributes

Before each CTL function is called, it must be supplied with values for its parameters. For an input parameter with name p, the value is found as follows:

```
if p is varying
  if the previous function has an output parameter with name p
    use previous function's output parameter
  else if the previous function has an output parameter whose name is p concatenated with “Out”
    use previous function's output parameter
  else if inFb contains a slice with name p
    use inFb slice
  else if input parameter p has a default value
```

use default value
else
error (throw `Iex::ArgExc`)
else (p is uniform)
if the previous function has an output parameter with name p
use previous function's output parameter
else if the previous function has an output parameter whose name is p concatenated with “Out”
use previous function's output parameter
else if `inHeader` contains an attribute with name p
use `inHeader` attribute
else if `envHeader` contains an attribute with name p
use `envHeader` attribute
else if input parameter p has a default value
use default value
else
error (throw `Iex::ArgExc`)

In all cases, the type of the value used must match the type of the input parameter. A type mismatch is an error; in this case `applyTransforms()` throws an `Iex::TypeExc` exception.

After each CTL function returns, the values of its output parameters may be copied into `outHeader` or `outFb`, in addition to potentially being used as inputs to the next CTL function. An output parameter with name p is handled as follows:

if p is varying
if `outFb` contains a slice with name p
copy the value into the `outFb` slice
else if name p ends in “Out” and `outFb` contains a slice whose name is p, except without the trailing “Out”
copy the value into the `outFb` slice

else (p is uniform)
if `outHeader` contains an attribute with name p
copy the value into the `outHeader` attribute
else if name p ends in “Out” and `outHeader` contains an attribute whose name is p, except without the trailing “Out”
copy the value into the `outHeader` attribute

The type of the output parameter must match the type of the frame buffer slice or header attribute. A type mismatch is an error; in this case `applyTransforms()` throws an `Iex::TypeExc` exception. `applyTransforms()` does not add attributes to `outHeader` or slices to the `outFb`. Only existing attributes or slices are used.

Example: `inFb` has two slices, `A` and `C`, and `outFb` has three slices, `AOut`, `B` and `C`. `transformNames` lists two transforms, `transform1` and `transform2`, with the following signatures:

```
void
transform1
    (varying input  half A,
     varying output half AOut,
     varying output half B)
{
    ...
}

void
transform2
    (varying input  half A,
     varying input  half B,
     varying input  half C,
     varying output half AOut,
     varying output half BOut,
     varying output half COut)
{
    ...
```

```
}
```

In this case, frame buffer slices and transform parameters are connected as follows:

Source	Destination
`inFb`, slice `A`	`transform1`, parameter `A`
`inFB`, slice `C`	`transform2`, parameter `C`
`transform1`, parameter `AOut`	`transform2`, parameter `A`
`transform1`, parameter `B`	`transform2`, parameter `B`
`transform2`, parameter `AOut`	`outFb`, slice `AOut`
`transform2`, parameter `BOut`	`outFb`, slice `B`
`transform2`, parameter `COut`	`outFb`, slice `C`

A.3 Translation between CTL Types, Channel Types and Attribute Types

Function `applyTransforms()` translates a fixed set of OpenEXR attribute and channel types to and from CTL, as shown in the table below. Parameter values of arbitrary type can be passed from one transform to the next, but only parameter values of the types listed in the table can be read from `inHeader`, `envHeader` or `inFb`, or stored in `outHeader` or `outFb`.

OpenEXR channel type	CTL function parameter type
`HALF`	`varying half`
`FLOAT`	`varying float`
`UINT`	`varying unsigned int`

OpenEXR attribute type	CTL function parameter type

`Box2iAttribute`	`uniform Box2i`, where `Box2i` is defined as
	`struct Box2i`
	`{`
	`  int min[2];`
	`  int max[2];`
	`};`

`Box2fAttribute`	`uniform Box2f`, where `Box2f` is defined as
	`struct Box2f`
	`{`
	`  float min[2];`
	`  float max[2];`
	`};`

`ChromaticitiesAttribute`	`uniform Chromaticities`, where `Chromaticities` is defined as
	`struct Chromaticities`
	`{`
	`  float red[2];`
	`  float green[2];`
	`  float blue[2];`
	`  float white[2];`
	`};`

OpenEXR attribute type	CTL function parameter type
`DoubleAttribute`	`uniform float`
`FloatAttribute`	`uniform float`
`IntAttribute`	`uniform int`
`M33fAttribute`	`uniform float[3][3]`
`M44fAttribute`	`uniform float[4][4]`
`V2iAttribute`	`uniform int[2]`
`V3fAttribute`	`uniform float[3]`
`V3iAttribute`	`uniform int[3]`

Appendix B Source Code

The C++ source code for the reference implementation of the CTL interpreter can be downloaded from

`http://sourceforge.net/projects/ampasctl`

The CTL interpreter depends on a set of low-level utility libraries, called IlmBase, which can be downloaded from

`http://savannah.nongnu.org/projects/openexr`

Appendix C Sample Code and Utilities

The OpenEXR_Viewers package, available at

```
http://savannah.nongnu.org/projects/openexr
```

includes source code for OpenEXR still image and moving image viewers, both with CTL support. In addition to the packages listed above, building the image viewers requires the OpenEXR and OpenEXR_CTL, packages, which are available at

```
http://savannah.nongnu.org/projects/openexr
```

and

```
http://sourceforge.net/projects/ampasctl
```

The source code packages mentioned contain instructions for building and installing the respective libraries and executables.