Blis: Better Language for Image Stuff
Final Report
Programming Languages and Translators, Spring 2017

Abbott, Connor (cwa2112)	`[System Architect]`
Pan, Wendy (wp2213)	`[Manager]`
Qinami, Klint (kq2129)	`[Language Guru]`
Vaccaro, Jason (jhv2111)	`[Tester]`

May 10, 2017

1 Introduction

Blis is a programming language for writing hardware-accelerated 3D rendering programs. It gives the programmer fine-grained control of the graphics pipeline while abstracting away the burdensome details of OpenGL. The Blis philosophy is that OpenGL provides more power and flexibility than some graphics programmers will ever need. These OpenGL programmers are forced to write boilerplate code and painfully long programs to accomplish the simplest of tasks. By exposing only the bare essentials of the graphics pipeline to the programmer, Blis decreases the number of decisions that a graphics developer has to make. The result is sleek, novice-friendly code.

With Blis, you can write real-time 3D rendering programs such as 3D model viewers. You can write shadow maps for rendering dynamic shadows and use render-to-texture techniques to produce a variety of effects. In short, the idea is that you can write programs that manipulate Blis' simplified graphics pipeline.

In particular, writing vertex and fragment shaders is now more convenient. Rather than having to write shader code in a separate shader language (GLSL), you can use Blis to write both tasks that run on the GPU and those that run on the CPU. Consequently, shaders can reuse code from other parts of the program. Uniforms, attributes, and textures are registered with shaders by simply passing them as arguments into user-defined shader functions. Furthermore, loading shaders from the CPU to GPU is easily accomplished by uploading to a built-in pipeline object. See the "Parts of the Language" section below for more information about this pipeline object, which is a central feature of Blis.

Blis has two backends: one for compiling source code to LLVM and another for compiling to GLSL. The generated LLVM code links to the OpenGL library in order to make calls that access the GPU.

2 Sorts of programs to be written

Rendering is a major subtopic of 3D computer graphics, with many active researchers developing novel techniques for light transport modeling. These researchers would like to test their new ray tracing, ray casting, and rasterization developments by actual experimentation. These new methods would be translated into rendering programs written in our language. Because these programs would be written at a higher level of abstraction than OpenGL, they would resemble mathematical thinking and derivations. Our language would also facilitate users to make larger tweaks to their programs with fewer lines of code, allowing for a more effective use of their time in experimentation. Minimizing this transaction cost between research and code should allow for more development of rendering software overall.

3 Language Tutorial

Below is a simple sample program for displaying a triangle. The language syntax is very similar to C. Use the @ symbol, e.g. @vertex and @fragment, to mark the entry points of GPU functions.

The pipelines, windows and buffers are declared as shown in the program. They are used the same way as in OpenGL.

```
@vertex vec4 vshader(vec3 pos)
{
  return vec4(pos.x, pos.y, pos.z, 1.);
}

@fragment void fshader(out vec3 color)
{
  color = vec3(1., 0., 1.);
}

pipeline my_pipeline {
  @vertex vshader;
  @fragment fshader;
};

int main()
{
  window win = window(1024, 768, false);

  set_active_window(win);

  buffer<vec3> b = buffer<vec3>();

  upload_buffer(b,
```

```
    vec3[3](vec3(-1., -1., 0.),
            vec3(1., -1., 0.),
            vec3(0., 1.,  0.)));

  pipeline my_pipeline p = pipeline my_pipeline(false);
  p.pos = b;

  while (true) {
    draw(p, 3);
    swap_buffers(win);
    poll_events();
    if (window_should_close(win))
      break;
  }

  return 0;
}
```

4 Language Reference Manual

4.1 Lexical Conventions

For the most part, tokens in Blis are similar to C. The main difference is the extra keywords added to support vector and matrix types. There are four kinds of tokens: identifiers, keywords, literals, and expression operators like `(`, `)`, and `*`. Blis is a free-format language, so comments and whitespace are ignored except to separate tokens.

4.1.1 Comments

Blis has C-style comments, beginning with `/*` and ending with `*/` which do not nest. C++ style comments, beginning with `//` and extending to the end of the line, are also supported.

4.1.2 Identifiers

An identifier consists of an alphabetic character or underscore followed by a sequence of letters, digits, and underscores.

4.1.3 Keywords

The following identifiers are reserved as keywords, and may not be used as identifiers:

- `if`
- `else`
- `for`
- `while`
- `return`
- `break`
- `continue`
- `int`
- `float`
- `u8`
- `bool`
- `window`
- `pipeline`
- `void`
- `struct`
- `const`
- `true`
- `false`
- `out`
- `inout`
- `uniform`
- `@gpu`
- `@gpuonly`
- `@vertex`
- `@fragment`

In addition, `vecN`, `ivecN`, `bvecN`, `u8vecN`, and `matNxM` are reserved, where `N` and `M` are any sequences of digits, although only `2`, `3`, and `4` are valid for `N` in `vecN`, `ivecN`, etc., and only `1` through `4` are valid for `N` and `M` in `matNxM`.

4.1.4 Literals

4.1.4.1 Integer Literals

Integer literals consist of a sequence of one or more of the digits 0-9. They will be interpreted as a decimal number.

4.1.4.2 Floating Point Literals

A floating-point literal consists of an integer part, a decimal, a fractional part, and `e` followed by an optionally signed exponent. Both the integer and fractional parts consist of a sequence of digits. At least one of the integer and fractional parts must be present, and at least one of the decimal point and the exponent must be present. Since Blis only supports single-precision floating point, all floating point constants are considered single-precision.

4.1.4.3 Character Literals

A character literal consists of a single character, or a backslash followed by one of `'`, `"`, `n`, `t`, or a sequence of decimal digits that must form a number from 0 to 255 which represents an ASCII code. `'\'`', `'\"'` `'\n'`, and `'\t'` have the usual meanings.

4.1.4.4 Boolean Literals

Boolean literals consist of the two keywords `true` and `false`.

4.1.4.5 String Literals

String literals consist of a series of characters surrounded by `"`. All the escapes described in section 4.1.4.3 are supported. String literals have type `u8[n]`, i.e. a fixed-size array of characters where `n` is the number of characters. For example, `"Hello world"` has type `u8[11]`. Unlike C, there is no extra `'\0'` inserted at the end of the string, since arrays are always sized anyways.

4.2 Types

$$
\begin{array}{lcc}
\textit{typ} & \rightarrow & \text{INT} \\
 & \rightarrow & \text{FLOAT} \\
 & \rightarrow & \text{BOOL} \\
 & \rightarrow & \text{BYTE} \\
 & \rightarrow & \text{STRUCT ID} \\
 & \rightarrow & \text{PIPELINE ID} \\
 & \rightarrow & \text{BUFFER} < \textit{typ} > \\
 & \rightarrow & \text{WINDOW} \\
 & \rightarrow & \text{VOID} \\
 & \rightarrow & \textit{typ}\ [\ ] \\
 & \rightarrow & \textit{typ}\ [\ \text{INT\_LITERAL}\ ]
\end{array}
$$

4.2.1 Integers

$\textit{typ} \rightarrow \text{INT}$

Blis supports integers through the `int` type, which is a 32-bit two's-complement signed integer type.

4.2.2 Characters

$\textit{typ} \rightarrow \text{BYTE}$

Blis supports characters through the `u8` type, which is an 8-bit unsigned integer type.

4.2.3 Floating Point

$\textit{typ} \rightarrow \text{FLOAT}$

Blis supports single-precision (32-bit) floating point numbers through the `float` type.

4.2.4 Booleans

$\textit{typ} \rightarrow \text{BOOL}$

Blis supports booleans through the `bool` type. Booleans may only ever be `true` or `false`.

4.2.5 Vectors and Matrices

typ $\rightarrow$ FLOAT

To more easily represent position data, color data, and linear transforms, Blis supports vectors of any built in type and matrices of floating-point numbers. Vectors are represented as `vecN` for floats, `ivecN` for integers, `bvecN` for booleans, and `u8vecN` for bytes/characters, where $2 \leq \texttt{N} \leq 4$. Matrices are represented by `matNxM` where $1 \leq \texttt{N} \leq 4$ and $1 \leq \texttt{M} \leq 4$. `N` is the number of columns and `M` is the number of rows, so that e.g. `mat1x1` is the same type as `float`, `mat1x2` is the same type as `vec2`, etc. Matrices are column-major, so a `mat2x3` logically contains two `vec3`'s which are interpreted as column vectors. Blis supports the full complement of matrix-vector multiplication operations as described in section 4.3.3, as well as component-wise multiplication on vectors and a number of built-in functions described in section 4.8.

All vector types have `x`, `y`, `z`, and `w` members as appropriate as if they were normal structures. Matrices `matNxM` where $\texttt{N} > 1$ also have `x`, `y`, `z`, and `w` members as appropriate, but they refer to the inner `N` `vecM`'s. Thus, the syntax to access the `x` component of a `vec4 foo` is `foo.x`, and the syntax to access the second column and first row of a `mat4x4 bar` is `bar.y.x`.

4.2.6 Arrays

typ $\rightarrow$ *typ* [ ]
$\rightarrow$ *typ* [INT_LITERAL]

Blis supports arrays of fixed and variable size. An array of `foo`'s is denoted as `foo[N]`, where `N` is an integer literal, or `foo[]` for a runtime-sized array. Arrays nest left-to-right, so that `int[3][2]` is an array of size 3 of an array of size 2 of integers.

4.2.7 Structures

typ $\rightarrow$ STRUCT ID
sdecl $\rightarrow$ STRUCT ID LBRACE *simple_vdecl_list* RBRACE SEMI
simple_vdecl_list $\rightarrow$ ϵ | *simple_vdecl_list* *typ* ID ;

Blis supports structures, similar to C, for storing multiple values together. Structures are declared with a syntax similar to C's, for example:

```
struct {
    vec3 a;
    float[3] b;
} MyStruct;

struct MyStruct makeMyStruct(int a, float[3] b);
```

However, declaring a struct and creating a variable of that type at the same time is *not* allowed. Thus, struct declarations must consist of the `struct` keyword, followed by an open brace, a list of `type name;` field declarations, a close brace, and then a semicolon.

4.2.8 Windows

$typ \rightarrow$ WINDOW

The `window` type is a builtin opaque type which represents a surface that represents a window from the window system that can be drawn to. The only way to interact with windows, other than constructing one as described in section 4.3.1 and drawing to the currently active window, is to use the builtin functions listed in section 4.8.

4.2.9 Pipelines

$typ \rightarrow$ PIPELINE ID

Blis programs can declare pipelines with a similar syntax to structures. Instead of declaring member fields, though, a pipeline declaration must specify the *vertex shader* and *fragment shader* entrypoints used through the `@vertex vertex_shader_name;` and `@fragment vertex_shader_name;` directives. Each directive must appear exactly once, although the order is arbitrary. Given a pipeline declaration `pipeline {...} my_pipeline;`, The vertex and fragment entrypoints are linked together at compile time to produce a pipeline type `pipeline my_pipeline`. Objects of type `pipeline my_pipeline` will contain certain fields that can be accessed as if the pipeline were a structure, as described in section 4.7.2. In addition, objects of any pipeline type can be used for drawing to a window with the `draw()` builtin, as described in section 4.8.

4.2.10 Buffers

$typ \rightarrow$ BUFFER $<$ typ $>$

Buffers represent arrays of data to be sent to the GPU. Since buffers can contain data of various types, they use a pseudo-template syntax `buffer<T>` where `T` can currently only be `float`, `int`, `vecN`, or `ivecN`. Buffers can be constructed as described in section 4.3.1, assigned to members of pipelines to bind them to the pipeline, and they can be filled with data using the `upload_buffer()` builtin as described in section 4.8.

4.3 Expressions

4.3.1 Type Constructors

$$\begin{array}{lcl} \textit{actuals\_opt} & \rightarrow & \epsilon \mid \textit{actuals\_list} \\ \textit{actuals\_list} & \rightarrow & \textit{expr} \mid \textit{actuals\_list} \text{ , } \textit{expr} \\ \textit{expr} & \rightarrow & \textit{typ} \text{ (} \textit{actuals\_opt} \text{)} \end{array}$$

Blis includes a consistent syntax for creating values of different types through *type constructors*. A type constructor is syntactically similar to a function call, except that instead of an identifier for the function to call there is a type. The arguments to the constructor are different for each type, and are described in the following sub-sections.

4.3.1.1 Vectors and Matrices

Constructors for vectors expect their components for arguments. For example, `ivec3(1, 2, 3)` constructs an `ivec3` whose components are 1, 2, and 3. Constructors for matrices accept their columns, so `mat3x2(vec2(1., 2.), vec2(3., 4.), vec2(5., 6.))` is a valid constructor for type `mat3x2`.

Constructors for integer, floating point, and boolean vectors as well as scalars can also accept another vector with the same number of components and one of the other two base types. This does the usual conversions on a per-component basis:

- Integer to float returns the closest floating-point number.
- Float to integer returns the closest integer.
- Boolean to float returns 0.0 for false and 1.0 for true.
- Boolean to integer returns 0 for false and 1 for true.
- Floating point and integer to boolean are equivalent to comparison with 0.0 and 0 respectively.

For example `vec2(ivec2(0, 1)) == vec2(0., 1.)`.

4.3.1.2 Arrays

Constructors for fixed-size arrays expect the same number of arguments as their size. For example, `vec4[10]()` expects 10 arguments of type `vec4`.

4.3.1.3 Structures

Constructors for structures have one argument for each member of the structure. They initialize each member of the structure. For example:

```
struct {
    int i;
    vec2 v;
} foo;

// ...

struct foo my_foo = struct foo(42, vec2(1., 2.));
// foo.i == 42, foo.v == vec2(1., 2.)
```

4.3.1.4 Windows

Constructors for window types accept two integers, describing the width and height of the window respectively, and a boolean argument which if true means that the window is created offscreen (for testing purposes).

4.3.1.5 Buffers and Pipelines

Constructors for buffers and pipelines take no arguments.

4.3.2 Function Calls

$$
\begin{aligned}
&\mathit{actuals\_opt} \;\rightarrow\; \epsilon \mid \mathit{actuals\_list} \\
&\mathit{actuals\_list} \;\rightarrow\; \mathit{expr} \mid \mathit{actuals\_list}\ ,\ \mathit{expr} \\
&\mathit{expr} \;\rightarrow\; \mathrm{ID}\ (\ \mathit{actuals\_opt}\ )
\end{aligned}
$$

Since Blis does not support function pointers, the function to call must simply be an identifier. Any call must be to a function defined on the top level, although it can be to a function defined later in the source. When matching function arguments, Blis first copies the actual arguments to any formal arguments marked as default or `inout` (see section **??**), so those must be implicitly assignable to the actuals as defined in section 4.3.4. Then, after the function call, it copies the formal arguments marked as `out` or `inout` to the actuals, so those formals must be implicitly assignable to the corresponding actuals and the actuals must be lvalues.

4.3.3 Operators

The operators supported by Blis, taken from GLSL, are:

Precendence	Operator Class	Operators	Associativity
1	Parenthetical Grouping	( )	Left to right
2	Array Subscript Function Call Structure Member Postfix Increment/Decrement	[ ] ( ) . ++, -	Left to right
3	Prefix Increment/Decrement Unary Operators	++, - -, !	Right to Left
4	Multiplicative Operators	*, /, %	Left to Right
5	Additive Binary Operators	+, -	Left to Right
6	Relational	<, >, <=, >=	Left to Right
7	Equality	==, !=	Left to Right
8	Logical And	&&	Left to Right
9	Logical Inclusive Or	||	Left to Right
10	Assignment	=	Right to Left

Evaluation proceeds left to right, and any side effects (for example, incrementing the lvalue by `++`) are evaluated immediately before returning the value.

4.3.3.1 Vectorized Operators

In addition to their normal operation on floating-point and boolean types, the following arithmetic operator classes can operate on vectors in a per-component manner:

- Unary Operators
- Multiplicative Operators
- Additive Binary Operators
- Logical And
- Logical Inclusive Or

That is, for example, if `a OP b` can take an `a` of type `float` and `b` of type `bool` and produce a result of type `float`, then it can also take an `a` of type `vec4` and `b` of type `bvec4` to produce a result of type `vec4`, where `result.x = a.x OP b.x`, `result.y = a.y OP b.y`, etc. In addition, if some of the inputs are of scalar types on the left-hand-side of a binary operation, then they will automatically be replicated to create a vector of the required size. Thus, the following is legal:

```
2.0 * vec3(1.0, 2.0, 3.0) // returns vec3(2.0, 4.0, 6.0)
```

but the following is not:

```
vec3(1.0, 2.0, 3.0) * vec2(1.0, 2.0)
```

In addition, all of the above operators except for `*` can operate per-component on matrices. `*` only works on matrices of floating-point type, and denotes linear algebraic matrix-matrix or matrix-vector multiplication. Multiplying a vector by a matrix is not permitted.

4.3.3.2 Relational Operators

Relational operators operate per component on vectors and return the logical and of all the per-component relations. The return type of relational checks is always a single boolean.

4.3.3.3 Equality

Equality (`==`) for vectors returns the logical and of all per component equality checks. Not equals (`!=`) returns the logical or of all per component not equals checks. In both cases, the return type is a single boolean.

4.3.3.4 Lvalues and Assignment

Expressions on the left-hand side of an `=` must be an *lvalue*, which must be either an identifier or one of the following:

- A member access using the `.` operator where the left-hand side is an lvalue.
- An array dereference using the `[ ]` operators where the left-hand side is an lvalue.
- An lvalue within parentheses.

When assigning a value, the right-hand side must be implicitly convertible to the left-hand side as defined in section 4.3.4.

4.3.4 Implicit Type Conversions

There are a few cases in Blis where one type can be an acceptable substitute for another, namely, sized vs. unsized arrays, and the type conversion happens implicitly. It is assumed that one type, known through the rest of the rules, needs to be converted to another type which is also known beforehand. In particular, implicit type conversions happen in the following places:

- For arguments when calling a function or implicit type constructor.
- For assignments and local variable initializations.
- For the return statements in functions with non-void return type.

A type A can be converted to another type B if:

- A and B are already the same type.
- A is of type `C[N]`, B is of type `D[]`, and C can be converted to D.

Note that if A and B are both struct or pipeline types, then for them to be the same type, they must have the same name; that is, structure equality is by name, not structural. Also, this relation is not symmetric (for example, `float[5]` can be converted to `float[]` but not vice versa), but it is transitive, and of course it is reflexive.

4.4 Statements and Control Flow

$$
\begin{array}{lll}
\textit{stmt\_list} & \rightarrow & \epsilon \mid \textit{stmt\_list}\ \textit{stmt} \\
\textit{stmt} & \rightarrow & \textit{expr}\ ; \\
 & \rightarrow & \textit{typ}\ \mathrm{ID}\ ; \\
 & \rightarrow & \textit{typ}\ \mathrm{ID} = \textit{expr}\ ; \\
 & \rightarrow & \{\ \textit{stmt\_list}\ \} \\
 & \rightarrow & \mathrm{BREAK}\ ; \\
 & \rightarrow & \mathrm{CONTINUE}\ ; \\
 & \rightarrow & \mathrm{RETURN}\ \textit{expr}\ ; \\
 & \rightarrow & \mathrm{IF}\ (\ \textit{expr}\ )\ \textit{stmt} \\
 & \rightarrow & \mathrm{IF}\ (\ \textit{expr}\ )\ \textit{stmt}\ \mathrm{ELSE}\ \textit{stmt} \\
 & \rightarrow & \mathrm{WHILE}\ (\ \textit{expr}\ )\ \textit{stmt} \\
 & \rightarrow & \mathrm{FOR}\ (\ \textit{expr}\ ;\ \textit{expr}\ ;\ \textit{expr}\ )\ \textit{stmt}
\end{array}
$$

A statement in Blis can be:

- A value followed by a semicolon.
- A variable declaration, which consists of the type, followed by an identifier, optionally followed by `=` and then an expression initializing the variable which must have a type implicitly convertible to the type of the variable.
- A *block* of statements surrounded by `{ }`.
- A `break` followed by a semicolon.
- A `continue` followed by a semicolon.
- A `return` followed by an optional value and a semicolon. The value must be implicitly convertable to the return type of the function, or empty if the function has a `void` return type.
- An `if` statement with optional else clause.

- A `for` loop.
- A `while` loop.

The syntax and semantics of `if`, `while`, and `for`, `break`, `continue`, and `return` are similar C. `return`, `break`, and `continue` must be the last statement in any block. When the function has a non-`void` return type, the type of the expression in the `return` must be implicitly convertible to the function return type as defined in section 4.3.4.

4.5 Top-level Constructs

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{decls}\ \$ \\
\textit{decls} & \rightarrow & \epsilon \\
& \rightarrow & \textit{decls vdecl} \\
& \rightarrow & \textit{decls fdecl} \\
& \rightarrow & \textit{decls sdecl} \\
& \rightarrow & \textit{decls pdecl}
\end{array}
$$

Programs in Blis consist of function definitions, structure declarations, pipeline declarations, and global variable declarations. Struct declarations are defined in section 4.2.7, and pipeline declarations are defined in section 4.2.9.

4.5.1 Global Variable Declarations

$$
\begin{array}{rcl}
\textit{vdecl} & \rightarrow & \textit{typ}\ \mathrm{ID}\ ; \\
& \rightarrow & \textit{typ}\ \mathrm{ID} = \textit{expr}\ ; \\
& \rightarrow & \mathrm{CONST}\ \textit{typ}\ \mathrm{ID} = \textit{expr}\ ;
\end{array}
$$

Global variable declarations consist of the type, the name, an optional initializer, and a semicolon. The initializer currently must be a literal as defined in section 4.1.4. When an initializer is present, the global variable will be initialized to that value before the program starts. In addition, a global variable may have a `const` qualifier, which means that references to it are not considered an lvalue. `const` global variables must have an initializer. No two global global variables can have the same name; that is, `const` and non-`const` global variables share the same namespace.

4.6 Function Definitions

$$
\begin{array}{lcl}
\textit{formal\_qualifier} & \rightarrow & \epsilon \mid \mathrm{OUT} \mid \mathrm{INOUT} \mid \mathrm{UNIFORM} \\
\textit{formal} & \rightarrow & \textit{formal\_qualifier}\ \textit{typ}\ \mathrm{ID} \\
\textit{formal\_list} & \rightarrow & \textit{formal} \mid \textit{formal\_list}\ ,\ \textit{formal} \\
\textit{formals\_opt} & \rightarrow & \epsilon \mid \textit{formals\_list} \\
\textit{func\_qualifier} & \rightarrow & \epsilon \mid \mathrm{GPUONLY} \mid \mathrm{VERTEX} \mid \mathrm{FRAGMENT} \mid \mathrm{GPU} \\
\textit{fdecl} & \rightarrow & \textit{func\_qualifier}\ \textit{typ}\ \mathrm{ID}\ (\ \textit{formals\_opt}\ )\ \{\ \textit{stmt\_list}\ \}
\end{array}
$$

Functions in Blis are declared similar to how they are in C:

```
// formal parameters must match
int my_cool_function(int a, vec3[3] b) {
    // ...
    return 42;
}
```

The body of a function consists of a block of statements as defined in the previous section.

Functions have qualifiers which indicate which context they may run in. The default, i.e. no qualifier, means that the function can only run on the CPU. The `@gpu` qualifier means that the function may run on *both* the GPU and CPU, and `@gpuonly` means that the function can only be run on the GPU. Finally, the `@vertex` and `@fragment` qualifiers mean that the function is a *vertex entrypoint* or *fragment entrypoint*. Such functions may not be called directly. Instead, they are included into pipelines as described in section 4.2.9, and then they are called by the GPU for each vertex or fragment to process. The following table summarizes which kinds of function can call which:

	caller: none	caller: `@gpu`	caller: `@gpuonly`	caller: `@vertex`/`@fragment`
callee: none	✓			
callee: `@gpu`	✓	✓	✓	✓
callee: `@gpuonly`			✓	✓
callee: `@vertex`/`@fragment`				

In addition, the call graph of functions with `@gpu` and `@gpuonly` qualifiers must not form a cycle; in other words, such functions cannot use recursion.

Formal parameters to functions may have an optional `out` or `inout` specifiers, similar to GLSL. At most one of `in` or `inout` can be added before the type of the formal parameter. Function parameters are always pass-by-value, but if `out` is specified, the reverse from the default happens: instead of the actual parameter being copied into the formal parameter at the beginning of the function, the formal parameter has an undefined value at the beginning of the function, and is copied to the actual parameter when returning from the function. `inout` is like `out`, except that the formal parameter is also initialized to the actual parameter at the beginning of the function, like normal parameters; in some sense, an `inout` parameter is a combination of the `out` and normal parameters. We follow GLSL in specifying that the order that these copies happen is undefined. In some sense, the behavior of `out` and `inout` is similar to pass-by-reference in that updates inside the function become visible outside, except that aliasing is not supported and so passing the same argument twice produces different results from what you would expect. For example, in this snippet:

```
void my_cool_function(inout int a, inout int b) {
    a = 2;
    // b still has value 1
    b = 3;
}

// ...

int c = 1;
my_cool_function(c, c);
// what is c now?
```

since the order in which `a` and `b` are copied to `c` after calling `my_cool_function` is undefined, `c` can be either 2 or 3 at the end.

Only vertex and fragment entrypoints may have formal parameters with a `uniform` qualifier, which is described in more detail in section 4.7.2.

4.6.1 Names and Scope

There exist the following namespaces in Blis:

- Variables (global and local).
- Structure members.
- Function names.
- Structure names.
- Pipeline names.

Structure names, structure members, pipeline names, function names, and global variable names are all simple flat namespaces where no two things may have the same name. However, names may be defined and used in any order; unlike in C, there is no restriction that a name is defined before it is used. The only restriction is that struct declarations cannot be cyclic; that is, a struct cannot contain fields of it own type, or any struct type that recursively contains fields of its type.

Similar to C, though, Blis has support for nested scopes for local variables. Global variables define the outer scope. The function arguments form an inner scope, and each block of statements also defines a new inner scope, which contains any bindings from the outer scope which aren't hidden by another binding from the inner scope. Inside these inner scopes, variables cannot be used before they are declared. Variable declarations always hide variable declarations from an outer scope and earlier variable declarations from an inner scope. Blis does not support separate linking, so there are no separate storage classes beyond local vs. global variables.

4.7 GPU specifics

4.7.1 Shader Entrypoints

To tell Blis that a particular function is meant to be compiled to a *shader* that executes on the GPU, it can be given a `@fragment` or `@vertex` annotations after the declaration but before the function body. Shader entrypoints are similar to normal functions, except that they can only call functions marked `@gpu` or `@gpuonly`. There are a few restrictions on functions called with these qualifiers:

- Recursion is disallowed. That is, the callgraph of functions marked with a qualifier cannot be cyclic.
- Runtime-sized arrays cannot be dereferenced or copied.
- No global variables can be referenced.

4.7.2 Pipelines

As described in section 4.2.9, pipelines can be declared that reference vertex shaders and fragment shaders. When a pipeline uses two shaders, they are checked for compatibility as follows:

- The types of `out` parameters of the vertex shader must match the types of `in` parameters of fragment shaders if they have the same name.
- The type of `uniform` parameters of vertex shader and fragment shader must match.
- The names of any `in` parameters to the vertex shader and `uniform` parameters of the fragment shader must not clash.

If the conditions are met, then each pipeline object is filled with the following members:

- The `in` parameters of the vertex shader, called *attributes*, are members of type `buffer<T>` if the input had type `T`. The attributes represent per-vertex data that is different for each invocation of the vertex shader.
- The `uniform` parameters of the vertex and fragment shaders are members of the same type.

4.8 Builtin Functions

- `@gpu float sin(float x)`: Computes the sine of the argument `x`.
- `@gpu float cos(float x)`: Computes the ccsine of the argument `x`.
- `@gpu float pow(float x, float y)`: Computes x^y.
- `@gpu float sqrt(float x)`: Computes $\sqrt{x}$.
- `@gpu float floor(float x)`: Computes $\lfloor x \rfloor$.
- `void print(u8[] s)`: Prints the string `s`.
- `void printi(int i)`: Prints the integer `i` followed by a newline.
- `void printb(bool b)`: Prints the boolean `b` followed by a newline.
- `void printf(float f)`: Prints the floating point number `f` followed by a newline.
- `void printc(char c)`: Prints the character `c` followed by a newline.
- `void set_active_window(window w)`: Changes the window used for drawing to `w`. Note that pipelines are implicitly tied to the window that was active when they were created, so if you switch from one window to another, any pipelines created under the previous window are invalid. Also, you can't create a pipeline until you call this function.
- `void swap_buffers(window w)`: Present whatever was drawn on the current window surface to the window system, and get a fresh buffer to draw on.
- `void poll_events()`: Ask the window system whether any events (mouse movements, key presses) have occured since the last time this function was called.
- `bool window_should_close(window w)`: Returns true if the user has pressed the close button on the window.
- `bool get_key(window w, int key)`: Returns true if the key associated with the given key code has been pressed.
- `bool get_mouse_button(window w, int button)`: Returns true if the given mouse button is pressed.
- `void get_mouse_pos(out float x, out float y)`: Return the x and y coordinates of the mouse.

- `vec4 read_pixel(int x, int y)`: Read the value of the pixel at the given x, y coordinates for the current window. Note that this will stall on any GPU drawing that occured beforehand.
- `u8[] read_file(u8[] path)`: Read in an entire file at the given path.
- `@gpu int length(T[] arr)`: Return the number of elements of a given array. Works for fixed-size and runtime-sized arrays.
- `void upload_buffer(buffer<T> buf, T[] data)`: replace the contents of `buf` with `data`.
- `void draw(pipeline _ foo, int verts)`: Given any pipeline `foo`, draw `verts` vertices to the current window.

In addition, there are a number of other builtins defined in Blis in `prelude.blis`. They include routines for parsing obj files, and building common kinds of transformation matrices.

5 Project Plan

5.1 Process used for planning, specification, development and testing

Our team communicated over Facebook messenger to coordinate who was going to work on what and to set deadlines for ourselves. We used Git as a version control system for our code and used ShareLatex and Overleaf to collaboratively edit LaTex documents. We met in-person to do pair programming and discuss ideas about the direction of the project.

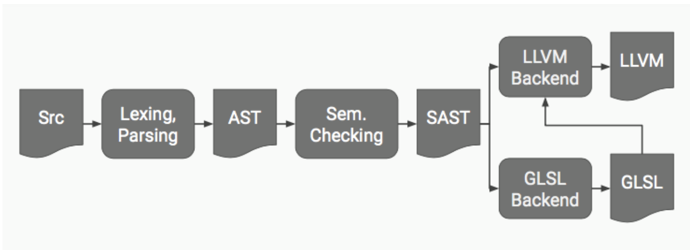
We used informative, concise commit messages so that everyone on the team could keep track of what features have been completed.

In our team, Connor is the system architect who is in charge of solving technical issues and designing language features. Wendy works as the project manager, setting the timeline for the project and handling logistics. Klint works as the language guru, making design decisions when we face disagreements. Jason is the tester, making sure each new feature is tested and that the tests are comprehensive.

Because our language has many similarities to C, the style paradigms used in Blis programming are very similar to those used in C. However, for those aspects of the language not in C, we adopted our own conventions. Some conventions that we adopted are as follows: vertex shaders are typically named vshader and fragment shaders are typically called fshader. If a function takes in "inout" or "out" arguments, the "in" arguments should be placed before the "inout" arguments which should be placed before the "out" arguments. Furthermore, when a function takes in "out" arguments as well as returns an int, this usually means that it returns 0 upon success and 1 if an error occurred.

6 Architectural Design

6.1 Diagram of Major Components



Expressions in the SAST are side effect free: assignments and function calls are statements in the SAST to make the two backends as similatr as possible.

All the group members worked on all parts of the compiler frontend. Connor, Wendy and Jason worked on the LLVM backend. Connor, Jason and Klint worked on the GLSL backend.

7 Test Plan

7.1 Representative Source Programs

7.1.1 Hello Triangle Source Code

```
@vertex vec4 vshader(vec3 pos)
{
  return vec4(pos.x, pos.y, pos.z, 1.);
}

@fragment void fshader(out vec3 color)
{
  color = vec3(1., 0., 1.);
}

pipeline my_pipeline {
  @vertex vshader;
  @fragment fshader;
};

int main()
{
  window win = window(1024, 768, false);

  set_active_window(win);

  buffer<vec3> b = buffer<vec3>();
```

```
  upload_buffer(b,
    vec3[3](vec3(-1., -1., 0.),
            vec3(1., -1., 0.),
            vec3(0., 1.,  0.)));

  pipeline my_pipeline p = pipeline my_pipeline(false);
  p.pos = b;

  while (true) {
    draw(p, 3);
    swap_buffers(win);
    poll_events();
    if (window_should_close(win))
      break;
  }

  return 0;
}
```

7.1.2 Obj Viewer Source Code

```
@vertex vec4 vshader(uniform mat4x4 xform, uniform mat4x4 world_xform,
                     vec3 pos, vec3 normal,
                     out vec3 world_pos, out vec3 world_normal)
{
  vec4 pos = vec4(pos.x, pos.y, pos.z, 1.);
  vec4 tmp = world_xform * pos;
  world_pos = vec3(tmp.x, tmp.y, tmp.z);
  tmp = world_xform * vec4(normal.x, normal.y, normal.z, 0.);
  world_normal = vec3(tmp.x, tmp.y, tmp.z);
  return xform * pos;
}

@fragment void fshader(vec3 world_pos,
                       vec3 world_normal,
                       uniform bool checkerboard,
                       uniform float checkerboard_size,
                       uniform vec3 light_pos,
                       uniform vec4 light_ambient,
                       uniform vec4 light_diffuse,
                       uniform vec4 light_specular,
                       uniform vec4 mat_ambient,
                       uniform vec4 mat_diffuse,
                       uniform vec4 mat_specular,
                       uniform float mat_shininess,
```

```
                      uniform vec3 eye_pos,
                      out vec4 color)
{
  vec4 diffuse;
  vec4 specular;
  if (checkerboard) {
    vec3 scaled_pos = world_pos * (1. / checkerboard_size);
    vec3 block = vec3(floor(scaled_pos.x), floor(scaled_pos.y),
        floor(scaled_pos.z));
    ivec3 block = ivec3(block);
    bool white = (block.x + block.y + block.z) % 2 == 0;
    if (white)
      diffuse = vec4(1.0, 1.0, 1.0, 1.0);
    else
      diffuse = vec4(0.0, 0.0, 0.0, 1.0);
    specular = vec4(1.0, 1.0, 1.0, 1.0);
  } else {
    diffuse = mat_diffuse;
    specular = mat_specular;
  }

  vec3 viewer_dir = normalize3(eye_pos - world_pos);
  vec3 light_dir = normalize3(light_pos - world_pos);
  vec3 half = normalize3(light_dir + viewer_dir);

  vec4 ambient_color = mat_ambient * light_ambient;

  float dd = dot3(light_dir, world_normal);
  if (dd < 0.)
    dd = 0.;

  vec4 diffuse_color = dd * light_diffuse * diffuse;

  dd = dot3(half, world_normal);
  vec4 specular_color;
  if (dd > 0.0)
    specular_color = pow(dd, mat_shininess) * light_specular * specular;
  else
    specular_color = vec4(0.0, 0.0, 0.0, 1.0);

  color = ambient_color + diffuse_color + specular_color;
}

vec3[] calc_smooth_normals(vec3[] verts, int[] tris)
{
  vec3[] normals = vec3[](length(verts));
  int i;
  for (i = 0; i < length(verts); i++) {
```

```
    normals[i] = vec3(0., 0., 0.);
  }

  for (i = 0; i < length(tris); i = i + 3) {
    vec3 a = verts[tris[i]];
    vec3 b = verts[tris[i + 1]];
    vec3 c = verts[tris[i + 2]];

    vec3 normal = normalize3(cross(b - a, c - a));
    normals[tris[i]] = normals[tris[i]] + normal;
    normals[tris[i + 1]] = normals[tris[i + 1]] + normal;
    normals[tris[i + 2]] = normals[tris[i + 2]] + normal;
  }

  for (i = 0; i < length(normals); i++)
    normals[i] = normalize3(normals[i]);

  return normals;
}

pipeline my_pipeline {
  @vertex vshader;
  @fragment fshader;
};

void set_transform(pipeline my_pipeline p, float theta, float phi, float
    distance)
{
  mat4x4 ctm = rotate_y(deg_to_rad(theta)) * rotate_z(deg_to_rad(phi));
  vec4 eye_point = ctm * vec4(distance, 0., 0., 1.);
  vec3 eye_point = vec3(eye_point.x, eye_point.y, eye_point.z);
  mat4x4 camera = look_at(eye_point, vec3(0., 0., 0.) /* at */,
                        vec3(0., 1., 0.) /* up */);
  mat4x4 proj = perspective(45. /* degrees */, 1., 0.5, 50.);
  p.eye_pos = eye_point;
  p.world_xform = identity();
  p.xform = proj * camera;
}

bool mouse_down = false;
bool x_pressed = false;
bool z_pressed = false;
bool c_pressed = false;
bool d_pressed = false;
float lastx;
float lasty;
const float MOUSE_SPEED = 1.; // degrees per pixel
bool increasing = false;
```

```
void update_camera_pos(window w, inout float theta, inout float phi,
                       inout float distance, inout bool checkerboard,
                       inout bool disco, inout vec4 color)
{
  if (get_mouse_button(w, MOUSE_BUTTON_LEFT)) {
    float x;
    float y;
    get_mouse_pos(w, x, y);
    if (mouse_down) {
      float diffx = x - lastx;
      float diffy = y - lasty;
      if (diffx != 0.) {
        theta = theta + MOUSE_SPEED * diffx;
        if (theta > 360.)
          theta = theta - 360.;
        if (theta < 0.)
          theta = theta + 360.;
      }
      if (diffy != 0.) {
        phi = phi + MOUSE_SPEED * diffy;
        if (phi > 85.)
          phi = 85.;
        if (phi < -85.)
          phi = -85.;
      }
    }
    lastx = x;
    lasty = y;
    mouse_down = true;
  } else {
    mouse_down = false;
  }

  if (get_key(w, KEY_X)) {
    if (!x_pressed) {
      distance = distance + 1.;
      if (distance > 50.)
        distance = 50.;
    }
    x_pressed = true;
  } else {
    x_pressed = false;
  }

  if (get_key(w, KEY_Z)) {
    if (!z_pressed) {
      distance = distance - 1.;
```

```
      if (distance < 1.)
        distance = 1.;
    }
    z_pressed = true;
  } else {
    z_pressed = false;
  }

  if (get_key(w, KEY_C)) {
    if (!c_pressed) {
      checkerboard = !checkerboard;
    }
    c_pressed = true;
  } else {
    c_pressed = false;
  }

  if (get_key(w, KEY_D)) {
    if (!d_pressed) {
      disco = !disco;
      if (disco)
        color = vec4(0., 0.5, 0.5, 1.);
    }
    d_pressed = true;
  } else {
    d_pressed = false;
  }

  if (disco) {
    if (increasing) {
      color.x = color.x + 0.05;
      if (color.x > 1.0) {
        color.x = 1.0;
        increasing = false;
      }
    } else {
      color.x = color.x - 0.05;
      if (color.x < 0.0) {
        color.x = 0.0;
        increasing = true;
      }
    }
  } else {
    color = vec4(1.0, 0.8, 0.0, 1.0);
  }
}

int main()
```

```
{
  vec3[] verts;
  int[] tris;

  if (!read_obj("./bunny_regular.obj", verts, tris)) {
    print("error reading obj file\n");
    return 1;
  }

  window win = window(1024, 768, false);

  set_active_window(win);

  buffer<vec3> b = buffer<vec3>();
  buffer<int> indices = buffer<int>();
  buffer<vec3> normals = buffer<vec3>();

  upload_buffer(b, verts);
  upload_buffer(indices, tris);
  upload_buffer(normals, calc_smooth_normals(verts, tris));

  pipeline my_pipeline p = pipeline my_pipeline(true);
  p.pos = b;
  p.normal = normals;
  p.indices = indices;

  p.light_pos = vec3(100., 100., 100.);
  p.light_ambient = vec4(0.2, 0.2, 0.2, 1.0);
  p.light_diffuse = vec4(1.0, 1.0, 1.0, 1.0);
  p.light_specular = vec4(1.0, 1.0, 1.0, 1.0);

  p.mat_ambient = vec4(1.0, 0.0, 1.0, 1.0);
  p.mat_specular = p.mat_diffuse;
  p.mat_shininess = 100.;

  p.checkerboard_size = 0.25;

  float distance = 2.;
  float theta = 0.;
  float phi = 0.;
  bool disco = false;

  while (true) {
    update_camera_pos(win, theta, phi, distance, p.checkerboard, disco,
        p.mat_diffuse);
    set_transform(p, theta, phi, distance);
    clear(vec4(1., 1., 1., 1.));
    draw(p, length(tris));
```

```
    swap_buffers(win);
    poll_events();
    if (window_should_close(win))
      break;
  }

  return 0;
}
```

7.2 Testing Suite and Justification

For testing, we practiced test-driven development so that we crafted tests as we added features.

Our test suite is separated into two groups: tests for the LLVM backend and tests for the GLSL backend. These involve success tests as well as fail tests. The success tests ensure that correct programs run as expected and fail tests check that the compiler is detecting errors and throwing the appropriate error messages when incorrect programs are written. We named fail tests files using the prefix "fail" and success files using the prefix "test" or "shadertest". Those beginning with "test" are success tests for the LLVM backend while those beginning with "shadertest" are success tests for the GLSL backend.

The tests for our LLVM backend check the code generation by comparing print statement outputs to expected output. We did not, however, use this comparison method to test our GLSL backend because shader code is run on the GPU and therefore cannot contain print statements. Instead, we have the fragment shader return a green pixel to indicate that the test has passed and a red pixel to indicate that the test has failed. The tests below can be found in our tests/ directory.

LLVM backend tests: (Contributions: Connor, Wendy, Klint)

```
test-add1.mc          test-conversions.mc    test-func2.mc     test-global4.mc       test-local-if.mc      test-matMult4.mc    test-specialFunc1.mc  test-var1.mc
test-arith1.mc        test-conversions2.mc   test-func3.mc     test-global5.mc       test-local-scope.mc   test-matMult5.mc    test-specialFunc2.mc  test-var2.mc
test-arith2.mc        test-fib.mc            test-func4.mc     test-hello.mc         test-local1.mc        test-matMult6.mc    test-specialFunc3.mc  test-vec1.mc
test-arith3.mc        test-float-arith1.mc   test-func5.mc     test-if1.mc           test-local2.mc        test-matUnop1.mc    test-specialFunc4.mc  test-vec2.mc
test-arith4.mc        test-float-arith2.mc   test-func6.mc     test-if2.mc           test-mat1.mc          test-modTest1.mc    test-split.mc         test-vec3.mc
test-array1.mc        test-float-arith3.mc   test-func7.mc     test-if3.mc           test-mat2.mc          test-ops1.mc        test-string1.mc       test-vec4.mc
test-array2.mc        test-float-literal.mc  test-func8.mc     test-if4.mc           test-matBinops1.mc    test-ops2.mc        test-struct1.mc       test-while1.mc
test-binop1.mc        test-for-break.mc      test-gcd.mc       test-if5.mc           test-matBinops2.mc    test-out.mc         test-struct2.mc       test-while2.mc
test-binop2.mc        test-for-continue.mc   test-gcd2.mc      test-incDec1.mc       test-matBinops3.mc    test-out2.mc        test-struct3.mc
test-char1.mc         test-for1.mc           test-global1.mc   test-initializer.mc   test-matMult1.mc      test-parse-obj.mc   test-struct4.mc
test-comments.mc      test-for2.mc           test-global2.mc   test-inout.mc         test-matMult2.mc      test-parse.mc       test-var-array.mc
test-const-global.mc  test-func1.mc          test-global3.mc   test-io.mc            test-matMult3.mc      test-reassign.mc    test-var-array2.mc
```

```
fail-assign1.mc          fail-deref1.mc                    fail-for5.mc          fail-func12.mc   fail-global3.mc          fail-lvalue2.mc
fail-assign2.mc          fail-deref2.mc                    fail-func-qual1.mc    fail-func2.mc    fail-global4.mc          fail-lvalue3.mc
fail-assign3.mc          fail-duplicate-struct-member.mc   fail-func-qual2.mc    fail-func3.mc    fail-global5.mc          fail-no-return.mc
fail-assign4.mc          fail-duplicate-struct.mc          fail-func-qual3.mc    fail-func4.mc    fail-global6.mc          fail-nomain.mc
fail-break1.mc           fail-expr1.mc                     fail-func-qual4.mc    fail-func5.mc    fail-gpu-recursive.mc    fail-return1.mc
fail-const-global.mc     fail-expr2.mc                     fail-func-qual5.mc    fail-func6.mc    fail-if1.mc              fail-return2.mc
fail-cyclic-struct1.mc   fail-expr3.mc                     fail-func-qual6.mc    fail-func7.mc    fail-if2.mc              fail-unknown-struct1.mc
fail-cyclic-struct2.mc   fail-for1.mc                      fail-func-qual7.mc    fail-func8.mc    fail-if3.mc              fail-while1.mc
fail-cyclic-struct3.mc   fail-for2.mc                      fail-func1.mc         fail-func9.mc    fail-initializer.mc      fail-while2.mc
fail-dead1.mc            fail-for3.mc                      fail-func10.mc        fail-global1.mc  fail-local-if.mc
fail-dead2.mc            fail-for4.mc                      fail-func11.mc        fail-global2.mc  fail-lvalue1.mc
```

GLSL backend tests: (Contributions: Connor, Jason, Klint)

```
shadertest-add1.mc
shadertest-arith1.mc
shadertest-arith2.mc
shadertest-arith3.mc
shadertest-array1.mc
shadertest-array2.mc
shadertest-comments.mc
shadertest-conversions.mc
shadertest-conversions2.mc
shadertest-float-arith1.mc
shadertest-float-arith2.mc
shadertest-float-arith3.mc
shadertest-float-literal.mc
shadertest-for1.mc
shadertest-func-void.mc
shadertest-func.mc
shadertest-func1.mc
shadertest-func2.mc
shadertest-func2.mc~
shadertest-func3.mc
shadertest-func6.mc
shadertest-gcd.mc
shadertest-gcd2.mc
shadertest-global.mc
shadertest-if5.mc
shadertest-inout.mc
shadertest-keyword.mc
shadertest-local-if.mc
shadertest-local-scope.mc
shadertest-local1.mc
shadertest-mat1.mc
shadertest-matBinop1.mc
shadertest-matMult1.mc
shadertest-matMult2.mc
shadertest-matMult3.mc
shadertest-matMult4.mc
shadertest-matMult5.mc
shadertest-modTest1.mc
shadertest-ops1.mc
shadertest-ops2.mc
shadertest-ops3.mc
shadertest-out.mc
shadertest-specialFunc1.mc
shadertest-specialFunc2.mc
shadertest-specialFunc3.mc
shadertest-specialFunc4.mc
shadertest-struct-keyword.mc
shadertest-struct1.mc
shadertest-trivial.mc
shadertest-var1.mc
shadertest-vec1.mc
shadertest-while1.mc
shadertest-while2.mc
shadertest-while3.mc
shadertest-while4.mc
```

7.3 Testing Automation and Scripts

We used Professor Edward's MicroC test script as the starting point for our test script and modified it to automate the tests for our GLSL backend.

8 Lessons Learned

8.1 Connor Abbott

I was a bit lucky, since I already had quite a bit of experience with compilers in general, and a little bit with LLVM. That made me unafraid of hacking on the compiler from an early stage. Yet still, there were a number of things that there simply wasn't time for. A compiler supplies essentially an infinite amount of work, but unfortunately we only have a finite amount of time, and in particular, a single semester isn't a lot of it. If I had to start again, I would have focused much earlier on getting the hello world (which for us was the triangle demo used in the tutorial) working earlier, rather than working on features like structures or advanced variable scoping.

8.2 Wendy Pan

Start early on the project. Writing a compiler is a long process, especially if you want to have all the features you want to implement in the proposal. Also, having great teammates speeds things up a great deal and being able to collaborate well with all teammates is probably the most important thing in completing the project.

8.3 Klint Qinami

Be ambitious with what you want to accomplish. There's no harm in setting out to do too much. Also, OCaml turns out to be a pretty great language for writing compilers. Automatic detection of missing cases in the case-matching system makes implementing new features much simpler: pretend it's already added, see what the compiler complains about, stop the complaints, and once it compiles, the feature will just work.

8.4 Jason Vaccaro

My advice for future students is that you should do research about this class before the start of the semester. Take a look at Edward's slides from previous semesters and check out previous students' projects. Start thinking about what kind of language you would be interested in implementing. You will be asked to begin designing your language right at the beginning of the semester, so it's not like there are several lectures dedicated towards helping you figure out how to design your language. You actually already know a lot about how to design a language before taking this course, so you can start thinking about your project before the semester begins. An important lesson I learned was that knowing how to write shell scripts is a useful tool for any programmer to have in their arsenal.

Blis Translator Code Listing

Final Report Appendix

Programming Languages and Translators, Spring 2017

Authors: Connor, Wendy, Klint, Jason

May 10, 2017

scanner.mll

```
(* Ocamllex scanner for MicroC *)

{ open Parser }

let digits = ['0'-'9']+
let exp = ['e''E'] ['+' '-']? digits

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }        (* Comments *)
| "//" [^'\n']* '\n' { token lexbuf } (* C++ style comments *)
| '('     { LPAREN }
| ')'     { RPAREN }
| '{'     { LBRACE }
| '}'     { RBRACE }
| '['     { LBRACKET }
| ']'     { RBRACKET }
| ';'     { SEMI }
| ','     { COMMA }
| '+'     { PLUS }
| '-'     { MINUS }
| "++"    { INC }
| "--"    { DEC }
| '*'     { TIMES }
| '/'     { DIVIDE }
| '%'     { MOD }
| '='     { ASSIGN }
| '.'     { DOT }
| "=="    { EQ }
| "!="    { NEQ }
```

```
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| "&&"    { AND }
| "||"    { OR }
| "!"     { NOT }
| "if"    { IF }
| "else"  { ELSE }
| "for"   { FOR }
| "while" { WHILE }
| "return" { RETURN }
| "break" { BREAK }
| "continue" { CONTINUE }
| "int"   { INT(1) }
| "float" { FLOAT(1, 1) }
| "vec" (digits as width) {
  let width = int_of_string width in
  if width < 2 || width > 4 then
    raise (Failure("vecN with N not between 2 and 4"))
  else
    FLOAT(1, width) }
| "ivec" (digits as width) {
  let width = int_of_string width in
  if width < 2 || width > 4 then
    raise (Failure("ivecN with N not between 2 and 4"))
  else
    INT(width) }
| "u8"    { BYTE(1) }
| "bool"  { BOOL(1) }
| "bvec" (digits as width) {
  let width = int_of_string width in
  if width < 2 || width > 4 then
    raise (Failure("bvecN with N not between 2 and 4"))
  else
    BOOL(width) }
| "u8vec" (digits as width) {
  let width = int_of_string width in
  if width < 2 || width > 4 then
    raise (Failure("u8vecN with N not between 2 and 4"))
  else
    BYTE(width) }
| "mat" (digits as w) "x" (digits as l) {
    let w = int_of_string w in
    let l = int_of_string l in
    if (w < 1 || w > 4) then
        raise (Failure("mat width not between 1 and 4"))
    else if (l < 1 || l > 4) then
```

```
      raise (Failure("mat␣length␣not␣between␣1␣and␣4"))
    else
      FLOAT(w, l) }
| "window" { WINDOW }
| "buffer" { BUFFER }
| "pipeline" { PIPELINE }
| "void"  { VOID }
| "struct" { STRUCT }
| "const" { CONST }
| "true"  { TRUE }
| "false" { FALSE }
| "out" { OUT }
| "inout" { INOUT }
| "uniform" { UNIFORM }
| digits as lxm { INT_LITERAL(int_of_string lxm) }
| (digits exp | (digits '.' digits? | '.' digits) exp?) as lxm
  { FLOAT_LITERAL(float_of_string lxm) }
| "@gpuonly" { GPUONLY }
| "@gpu"     { GPU }
| "@vertex"  { VERTEX }
| "@fragment" { FRAGMENT }
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| "'" ([^ '\'' '\\'] as c) "'" { CHAR_LITERAL(c) }
| "'\\n'" { CHAR_LITERAL('\n') }
| "'\\t'" { CHAR_LITERAL('\t') }
| "'\\''" { CHAR_LITERAL('\'') }
| "'\\\"'" { CHAR_LITERAL('"') }
| "\\\\" { CHAR_LITERAL('\\') }
| "'\\" (digits as d) "'" {
  let value = int_of_string d in
  if value > 255 then
    raise (Failure "character␣escape␣must␣be␣0-255")
  else
    CHAR_LITERAL(Char.chr value)
}
| '"' { STRING_LITERAL(str "" lexbuf) }
| eof { EOF }
| _ as char { raise (Failure("illegal␣character␣" ^ Char.escaped char)) }

and str old_str = parse
[^ '\n' '"' '\\']+ as c { str (old_str ^ c) lexbuf }
| "\\n" { str (old_str ^ "\n") lexbuf }
| "\\t" { str (old_str ^ "\t") lexbuf }
| "\\\"" { str (old_str ^ "\"") lexbuf }
| "\\'" { str (old_str ^ "\'") lexbuf }
| "\\" (digits as d) {
  let value = int_of_string d in
  if value > 255 then
```

```
    raise (Failure "character escape must be 0-255")
  else
    str (old_str ^ String.make 1 (Char.chr value)) lexbuf
}
| "\\\\" { str (old_str ^ "\\" ) lexbuf }
| "\\\n" { str (old_str ^ "\n") lexbuf }
| '"' { old_str }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char ^
  " in string literal")) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

semant.ml

```
(* Semantic checking for the MicroC compiler *)

open Ast
open Sast
open Utils

module StringMap = Map.Make(String)
module StringSet = Set.Make(String)

type symbol_kind =
    KindLocal
  | KindGlobal
  | KindConstGlobal

type symbol = symbol_kind * Ast.typ * string

type translation_environment = {
  scope : symbol StringMap.t;
  names : StringSet.t;
  locals : bind list;
  cur_qualifier : func_qualifier;
  in_loop : bool;
}

(* Semantic checking of a program. Returns the SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check program =
  let globals = program.var_decls in
  let functions = program.func_decls in
```

```
  let structs = program.struct_decls in
  let pipelines = program.pipeline_decls in

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
    n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  let find_symbol_table table name =
    if StringMap.mem name table then
      StringMap.find name table
    else
      raise (Failure ("undeclared␣identifier␣" ^ name))
  in

  (* if the name already exists, add a number to it to make it unique *)
  let find_unique_name env name =
    if not (StringSet.mem name env.names) then
      name
    else
      let rec find_unique_name'␣env␣name␣n␣=
␣␣␣␣␣␣␣␣let␣unique_name␣=␣name␣^␣string_of_int␣n␣in
␣␣␣␣␣␣␣␣if␣not␣(StringSet.mem␣unique_name␣env.names)␣then
␣␣␣␣␣␣␣␣␣␣unique_name
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣find_unique_name' env name (n + 1)
      in
      find_unique_name'␣env␣name␣1
␣␣in
␣␣(*␣return␣a␣fresh,␣never-used␣name,␣but␣don't actually add it to the
    symbol
   * table. Useful for creating compiler temporaries.
   *)
  let add_private_name env =
    let unique_name = find_unique_name env "_" in
    ({ env with names = StringSet.add unique_name env.names }, unique_name)
  in
  (* Adds/replaces symbol on the symbol table, returns the new unique name
      for
   * the symbol
   *)
  let add_symbol_table env name kind typ =
    let unique_name = find_unique_name env name in
    ({ env with scope = StringMap.add name (kind, typ, unique_name)
```

```
      env.scope;
     names = StringSet.add unique_name env.names; }, unique_name)
in

(**** Checking Structure and Pipeline Declarations ****)

report_duplicate
  (fun n -> "duplicate structure " ^ n)
  (List.map (fun s -> s.sname) structs);

report_duplicate
  (fun n -> "duplicate pipeline " ^ n)
  (List.map (fun p -> p.pname) pipelines);

let struct_decls = List.fold_left (fun m s ->
  StringMap.add s.sname s m) StringMap.empty structs
in

let pipeline_decls = List.fold_left (fun m p ->
  StringMap.add p.pname p m) StringMap.empty pipelines
in

let check_buffer_type = function
    Mat(Float, 1, _) | Mat(Int, 1, _) -> ()
  | _ as t -> raise (Failure ("bad type " ^ string_of_typ t ^ " for 
      buffer"))
in

let check_return_type exceptf = function
    (Struct s) -> if not (StringMap.mem s struct_decls) then
        raise (Failure (exceptf ("struct " ^ s)))
      else
        ()
  | (Pipeline p) -> if not (StringMap.mem p pipeline_decls) then
        raise (Failure (exceptf ("pipeline " ^ p)))
      else
        ()
  | (Buffer t) -> check_buffer_type t
  | _ -> ()
in

(* Raise an exception if a given binding is to a void type or a struct
    that
 * doesn't exist *)
  let rec check_type bad_struct void = function
      (Struct s, n) -> if not (StringMap.mem s struct_decls) then
          raise (Failure (bad_struct ("struct " ^ s) n))
        else
```

```
␣␣␣␣␣␣␣␣␣␣()
␣␣␣␣|␣(Pipeline␣p,␣n)␣->␣if␣not␣(StringMap.mem␣p␣pipeline_decls)␣then
␣␣␣␣␣␣␣␣␣␣raise␣(Failure␣(bad_struct␣("pipeline␣"␣^␣p)␣n))
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣()
␣␣␣␣|␣(Array(t,␣_),␣n)␣->␣check_type␣bad_struct␣void␣(t,␣n)
␣␣␣␣|␣(Buffer(t),␣_)␣->␣check_buffer_type␣t
␣␣␣␣|␣(Void,␣n)␣->␣raise␣(Failure␣(void␣n))
␣␣␣␣|␣_␣->␣()
␣␣in

␣␣List.iter␣(fun␣s␣->
␣␣␣␣report_duplicate
␣␣␣␣␣␣(fun␣m␣->␣"duplicate␣member␣"␣^␣m␣^␣"␣of␣structure␣"␣^␣s.sname)
␣␣␣␣␣␣(List.map␣snd␣s.members))␣structs;

␣␣List.iter␣(fun␣s␣->
␣␣␣␣List.iter␣(fun␣m␣->␣check_type
␣␣␣␣␣␣(fun␣sn␣n␣->␣sn␣^␣"␣does␣not␣exist␣in␣member␣"␣^␣n␣^
␣␣␣␣␣␣␣␣"␣of␣struct␣"␣^␣s.sname)
␣␣␣␣␣␣(fun␣n␣->␣"illegal␣void␣member␣"␣^␣n␣^␣"␣of␣struct␣"␣^␣s.sname)
␣␣␣␣␣␣m)
␣␣␣␣s.members)
␣␣structs;

␣␣(*␣sort␣structures␣and␣check␣for␣cycles␣in␣the␣definitions␣*)
␣␣let␣structs␣=
␣␣␣␣(*␣function␣from␣struct␣name␣to␣a␣list␣of␣struct␣names␣used␣in␣its␣
    members␣*)
␣␣␣␣let␣succs␣s␣=␣List.fold_left␣(fun␣succs␣m␣->
␣␣␣␣␣␣match␣base_type␣(fst␣m)␣with
␣␣␣␣␣␣␣␣␣␣Struct␣name␣->␣StringMap.find␣name␣struct_decls␣::␣succs
␣␣␣␣␣␣␣␣|␣_␣->␣succs)
␣␣␣␣[]␣s.members
␣␣␣␣in
␣␣␣␣tsort␣structs␣succs␣(fun␣cycle␣->
␣␣␣␣␣␣raise␣(Failure␣("cycle␣in␣struct␣definitions:␣"␣^
␣␣␣␣␣␣␣␣String.concat␣"␣->␣"␣(List.map␣(fun␣s␣->␣s.sname)␣cycle)))
␣␣in

␣␣(****␣Checking␣Global␣Variables␣****)

␣␣List.iter␣(check_type
␣␣␣␣(fun␣s␣n␣->␣s␣^␣"␣does␣not␣exist␣for␣global␣"␣^␣n)
␣␣␣␣(fun␣n␣->␣"illegal␣void␣global␣"␣^␣n))
␣␣(List.map␣(fun␣(_,␣b,␣_)␣->␣b)␣globals);

␣␣report_duplicate␣(fun␣n␣->␣"duplicate␣global␣"␣^␣n)
```

```
␣␣␣␣(List.map␣(fun␣(_,␣(_,␣n),␣_)␣->␣n)␣globals);

␣␣let␣env␣=␣{
␣␣␣␣in_loop␣=␣false;
␣␣␣␣scope␣=␣StringMap.empty;
␣␣␣␣cur_qualifier␣=␣Both;
␣␣␣␣locals␣=␣[];
␣␣␣␣names␣=␣StringSet.empty;␣}␣in

␣␣let␣check_const␣=␣function
␣␣␣␣(*␣TODO␣if␣we␣want␣to␣do␣more,␣we␣need␣to␣figure␣out␣how␣to␣pull␣out␣
    some␣of
␣␣␣␣␣*␣expr␣and␣re-use␣it␣here.
␣␣␣␣␣*)
␣␣␣␣␣␣IntLit(l)␣->␣(Mat(Int,␣1,␣1),␣SIntLit(l))
␣␣␣␣|␣FloatLit(l)␣->␣(Mat(Float,␣1,␣1),␣SFloatLit(l))
␣␣␣␣|␣BoolLit(l)␣->␣(Mat(Bool,␣1,␣1),␣SBoolLit(l))
␣␣␣␣|␣CharLit(c)␣->␣(Mat(Byte,␣1,␣1),␣SCharLit(c))
␣␣␣␣|␣StringLit(s)␣->
␣␣␣␣␣␣␣␣(Array(Mat(Byte,␣1,␣1),␣Some␣(String.length␣s)),␣SStringLit(s))
␣␣␣␣|␣_␣as␣e␣->␣raise␣(Failure␣("invalid␣expression␣"␣^␣string_of_expr␣e␣^
␣␣␣␣␣␣␣␣"␣in␣global␣variable␣initializer"))
␣␣in

␣␣let␣check_initializer␣(typ,␣name)␣e␣=
␣␣␣␣let␣se␣=␣check_const␣e␣in
␣␣␣␣if␣fst␣se␣<>␣typ␣then
␣␣␣␣␣␣raise␣(Failure␣("invalid␣type␣in␣initializer␣for␣"␣^␣name␣^␣",␣
    expected␣"␣^
␣␣␣␣␣␣␣␣string_of_typ␣typ␣^␣"␣but␣got␣"␣^␣string_of_typ␣(fst␣se)))
␣␣␣␣else
␣␣␣␣␣␣se
␣␣in

␣␣let␣env,␣globals␣=␣List.fold_left␣(fun␣(env,␣globals)␣(qual,␣(typ,␣
    name),␣init)␣->
␣␣␣␣let␣env,␣name␣=␣add_symbol_table␣env␣name
␣␣␣␣␣␣(if␣qual␣=␣GVConst␣then␣KindConstGlobal␣else␣KindGlobal)
␣␣␣␣typ␣in
␣␣␣␣env,␣(qual,␣(typ,␣name),␣(match␣init␣with
␣␣␣␣␣␣␣␣None␣->␣None
␣␣␣␣␣␣|␣Some␣e␣->␣Some␣(check_initializer␣(typ,␣name)␣e)))␣::␣globals)
␣␣(env,␣[])␣globals␣in

␣␣(****␣Checking␣Functions␣****)

␣␣report_duplicate␣(fun␣n␣->␣"duplicate␣function␣"␣^␣n)
␣␣␣␣(List.map␣(fun␣fd␣->␣fd.fname)␣functions);
```

```
␣␣(*␣Function␣declaration␣for␣a␣named␣function␣*)

␣␣let␣built_in_decls␣=
␣␣␣␣let␣int1␣=␣Mat(Int,␣1,␣1)␣and␣bool1␣=␣Mat(Bool,␣1,␣1)␣and␣float1␣=
␣␣␣␣␣␣␣␣Mat(Float,␣1,␣1)
␣␣␣␣and␣byte1␣=␣Mat(Byte,␣1,␣1)␣in␣[
␣␣␣␣␣{␣typ␣=␣float1;␣fname␣=␣"sin";␣formals␣=␣[In,␣(float1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣Both;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣float1;␣fname␣=␣"cos";␣formals␣=␣[In,␣(float1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣Both;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣float1;␣fname␣=␣"pow";␣formals␣=␣[In,␣(float1,␣"x");␣In,
␣␣␣␣␣␣(float1,␣"y")];␣fqual␣=␣Both;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣float1;␣fname␣=␣"sqrt";␣formals␣=␣[In,␣(float1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣Both;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣float1;␣fname␣=␣"floor";␣formals␣=␣[In,␣(float1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣Both;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"print";␣formals␣=␣[In,␣(Array(byte1,␣None),␣
    "x")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"printi";␣formals␣=␣[In,␣(int1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"printb";␣formals␣=␣[In,␣(bool1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"printf";␣formals␣=␣[In,␣(float1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"printc";␣formals␣=␣[In,␣(byte1,␣"x")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"set_active_window";␣formals␣=␣[In,␣(Window,␣
    "w")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"swap_buffers";␣formals␣=␣[In,␣(Window,␣"w")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"poll_events";␣formals␣=␣[];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣bool1;␣fname␣=␣"window_should_close";
␣␣␣␣␣␣␣formals␣=␣[In,␣(Window,␣"w")];␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣bool1;␣fname␣=␣"get_key";
␣␣␣␣␣␣␣formals␣=␣[In,␣(Window,␣"w");␣In,␣(int1,␣"key")];␣fqual␣=␣CpuOnly;
␣␣␣␣␣␣␣body␣=␣[];␣};
␣␣␣␣␣{␣typ␣=␣bool1;␣fname␣=␣"get_mouse_button";
␣␣␣␣␣␣␣formals␣=␣[In,␣(Window,␣"w");␣In,␣(int1,␣"button")];␣fqual␣=␣
    CpuOnly;
␣␣␣␣␣␣␣body␣=␣[];␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"get_mouse_pos";
␣␣␣␣␣␣␣formals␣=␣[In,␣(Window,␣"w");␣Out,␣(float1,␣"x");␣Out,␣(float1,␣
    "y")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[];␣};
```

```
␣␣␣␣␣{␣typ␣=␣Mat(Float,␣1,␣4);␣fname␣=␣"read_pixel";
␣␣␣␣␣␣␣formals␣=␣[In,␣(int1,␣"x");␣In,␣(int1,␣"y")];
␣␣␣␣␣␣␣fqual␣=␣CpuOnly;␣body␣=␣[]␣};
␣␣␣␣␣{␣typ␣=␣Array(byte1,␣None);␣fname␣=␣"read_file";
␣␣␣␣␣␣␣formals␣=␣[In,␣(Array(byte1,␣None),␣"file")];␣fqual␣=␣CpuOnly;␣body␣
    =␣[]␣};
␣␣␣␣␣{␣typ␣=␣Void;␣fname␣=␣"clear";
␣␣␣␣␣␣␣formals␣=␣[In,␣(Mat(Float,␣1,␣4),␣"color")];␣fqual␣=␣CpuOnly;␣body␣
    =␣[]
␣␣␣␣␣};
␣␣␣␣␣(*␣these␣builtins␣have␣type-checking␣rules␣not␣captured␣through␣the␣
    normal
␣␣␣␣␣␣*␣mechanism,␣so␣they␣are␣special-cased␣below.␣Add␣them␣here␣to␣make␣
    sure
␣␣␣␣␣␣*␣the␣program␣doesn't declare another function with the same name;
     * the types and qualifiers are ignored.
     *)
    { typ = Void; fname = "length"; formals = []; fqual = Both; body = []
        };
    { typ = Void; fname = "upload_buffer"; formals = []; fqual = CpuOnly;
      body = [] };
    { typ = Void; fname = "draw"; formals = []; fqual = CpuOnly;
      body = [] };
  ]
in

List.iter (fun built_in_decl ->
  let name = built_in_decl.fname in
  if List.mem name (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function␣" ^ name ^ "␣may␣not␣be␣defined")) else
      ())
built_in_decls;

let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname
    fd m)
                      StringMap.empty (built_in_decls @ functions)
in

let function_decl s = try StringMap.find s function_decls
     with Not_found -> raise (Failure ("unrecognized␣function␣" ^ s))
in

let main = function_decl "main" in (* Ensure "main" is defined *)
if main.fqual <> CpuOnly then
  raise (Failure "main␣function␣has␣bad␣qualifier")
else
  ()
;
```

```
(* Do compile-time checks for consistency of pipeline declarations.
 * We want to do this after function_decls has been constructed.
 *)

let pipelines = List.map (fun pd ->
  if pd.fshader = "" then
    raise (Failure ("pipeline␣" ^ pd.pname ^
      "␣doesn't␣contain␣a␣fragment␣shader"))
  else if pd.vshader = "" then
    raise (Failure ("pipeline␣" ^ pd.pname ^
      "␣doesn't␣contain␣a␣vertex␣shader"))
  else
    let vert_decl = function_decl pd.vshader in
    let frag_decl = function_decl pd.fshader in
    if vert_decl.fqual <> Vertex then
      raise (Failure
       ("vertex␣entrypoint␣" ^ pd.vshader ^ "␣in␣pipeline␣" ^ pd.pname ^
        "is␣not␣marked␣@vertex"))
    else if frag_decl.fqual <> Fragment then
      raise (Failure
       ("fragment␣entrypoint␣" ^ pd.vshader ^ "␣in␣pipeline␣" ^ pd.pname ^
        "is␣not␣marked␣@fragment"))
    else
      let decl_uniforms decl =
        List.fold_left (fun map (qual, (typ, name)) ->
          if qual <> Uniform then map else
            StringMap.add name typ map) StringMap.empty decl.formals
      in
      let (vuniforms : typ StringMap.t) = decl_uniforms vert_decl in
      let (funiforms : typ StringMap.t) = decl_uniforms frag_decl in
      let uniforms = StringMap.merge (fun name vtyp ftyp -> match (vtyp,
          ftyp) with
          (None, None) -> None
        | (Some typ, None) -> Some typ
        | (None, Some typ) -> Some typ
        | (Some vtyp',␣Some␣ftyp') -> if vtyp'␣<>␣ftyp' then raise
            (Failure (
           "differing␣types␣" ^ string_of_typ vtyp'␣^␣"␣and␣"␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣string_of_typ␣ftyp' ^ "␣for␣uniform␣" ^ name ^ "␣in␣pipeline␣
    " ^
           pd.pname))
          else Some vtyp')␣vuniforms␣funiforms␣in
␣␣␣␣␣␣␣␣let␣uniforms_list␣=␣List.map␣(fun␣(name,␣typ)␣->␣(typ,␣name))
␣␣␣␣␣␣␣␣␣␣(StringMap.bindings␣uniforms)␣in
␣␣␣␣␣␣␣␣let␣inputs_list␣=␣List.map␣(fun␣(_,␣(t,␣n))␣->␣(Buffer(t),␣n))
␣␣␣␣␣␣␣␣␣␣(List.filter␣(fun␣(qual,␣_)␣->␣qual␣=␣In)␣vert_decl.formals)␣in
␣␣␣␣␣␣␣␣let␣pipeline_members␣=
```

```
          (Buffer(Mat(Int, 1, 1)), "indices") :: uniforms_list @ inputs_list in
        report_duplicate (fun n -> "duplicate member " ^ n ^ " of pipeline " ^
          pd.pname) (List.map snd pipeline_members);
        { spname = pd.pname;
          sfshader = pd.fshader;
          svshader = pd.vshader;
          sinputs = inputs_list;
          suniforms = uniforms_list;
          smembers = pipeline_members;
        })
  pipelines
  in

  let pipeline_decls = List.fold_left (fun m p ->
    StringMap.add p.spname p m) StringMap.empty pipelines
  in

  let check_function func =

    List.iter (check_type
      (fun s n -> s ^ " does not exist for formal " ^ n ^ " in " ^
        func.fname)
      (fun n -> "illegal void formal " ^ n ^ " in " ^ func.fname))
    (List.map snd func.formals);

    check_return_type
      (fun s -> s ^ " does not exist in return type of function " ^
      func.fname) func.typ;

    (* checks related to uniform qualifiers *)
    List.iter (fun (qual, (typ, name)) ->
      if qual = Uniform then
        if func.fqual != Vertex && func.fqual != Fragment then
          raise (Failure ("uniform argument " ^ name ^ " declared in " ^
            func.fname ^ " which is not an entrypoint"))
        else
          match typ with
              Mat(Float, _, _) -> ()
            | Mat(Int, 1, _) -> ()
            | Mat(Bool, 1, _) -> ()
            | _ -> raise (Failure ("illegal type " ^ string_of_typ typ ^
              " used in a uniform argument in " ^ func.fname)))
    func.formals;

    report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
```

```
␣␣␣␣␣␣(List.map␣(fun␣(_,␣(_,␣n))␣->␣n)␣func.formals);

␣␣␣␣let␣check_call_qualifiers␣env␣fname␣fqual␣=
␣␣␣␣␣␣(match␣env.cur_qualifier,␣fqual␣with
␣␣␣␣␣␣␣␣␣␣(CpuOnly,␣CpuOnly)
␣␣␣␣␣␣␣␣|␣(CpuOnly,␣Both)
␣␣␣␣␣␣␣␣|␣(Vertex,␣GpuOnly)
␣␣␣␣␣␣␣␣|␣(Vertex,␣Both)
␣␣␣␣␣␣␣␣|␣(Fragment,␣GpuOnly)
␣␣␣␣␣␣␣␣|␣(Fragment,␣Both)
␣␣␣␣␣␣␣␣|␣(GpuOnly,␣GpuOnly)
␣␣␣␣␣␣␣␣|␣(GpuOnly,␣Both)
␣␣␣␣␣␣␣␣|␣(Both,␣Both)␣->␣()
␣␣␣␣␣␣␣␣|␣␣_␣->␣raise␣(Failure␣("cannot␣call␣"␣^␣string_of_func_qual␣fqual␣
    ^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"␣function␣"␣^␣fname␣^␣"␣from␣"␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣string_of_func_qual␣env.cur_qualifier␣^␣"␣function␣"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^␣func.fname))
␣␣␣␣in

␣␣␣␣(*␣create␣a␣new␣compiler␣temporary␣variable␣of␣the␣given␣type␣*)
␣␣␣␣let␣add_tmp␣env␣typ␣=
␣␣␣␣␣␣let␣env,␣name␣=␣add_private_name␣env␣in
␣␣␣␣␣␣({␣env␣with␣locals␣=␣(typ,␣name)␣::␣env.locals␣},␣(typ,␣SId(name)))␣
    in

␣␣␣␣let␣rec␣check_assign␣env␣lval␣rval␣stmts␣fail␣=
␣␣␣␣␣␣let␣ltyp␣=␣fst␣lval␣and␣rtyp␣=␣fst␣rval␣in
␣␣␣␣␣␣let␣array_assign␣t␣=
␣␣␣␣␣␣␣␣let␣int1␣=␣Mat(Int,␣1,␣1)␣and␣bool1␣=␣Mat(Bool,␣1,␣1)␣in
␣␣␣␣␣␣␣␣match␣snd␣rval␣with
␣␣␣␣␣␣␣␣␣␣␣␣STypeCons(_)␣when␣ltyp␣=␣rtyp␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣in␣this␣special␣case,␣we're just allocating a new array,
    and
              * copying the RHS in a loop would mean repeatedly allocating
              * memory -- which is silly. We know that the types match
                exactly,
              * and there are no aliasing concerns since no other name
                exists
              * for this chunk of memory. Therefore we just do a simple
              * assignment and move on.
              *)
             env, SAssign(lval, rval) :: stmts
         | _ ->
           let env, index = add_tmp env int1 in
           let stmts = SAssign(index, (int1, SIntLit(0))) :: stmts in
           let env, length = add_tmp env int1 in
           let stmts = SCall(length, "length", [rval]) :: stmts in
```

```
          let stmts = if ltyp = Array(t, None) then
            SAssign(lval, (ltyp, STypeCons([length]))) :: stmts else stmts
                in
          let env, stmt' = check_assign env (int1, SArrayDeref(lval, 
              index))
              (int1, SArrayDeref(rval, index)) [] fail in
            env, SLoop(
              SIf((bool1, SBinop(length, ILeq, index)), [SBreak], []) ::
                List.rev stmt',
            [SAssign(index, (int1, SBinop(index, IAdd, (int1,
                SIntLit(1)))))])
          :: stmts in
      match (ltyp, rtyp) with
        (Array(t, i), Array(_, i')) when i = i' -> array_assign t
      | (Array(t, Some _), Array(_, None)) | (Array(t, None), Array(_, Some
          _)) ->
        array_assign t
      | (Struct(s), Struct(s')) when s = s' ->
        let sdecl = StringMap.find s struct_decls in
        List.fold_left (fun (env, stmts) (t, n) ->
          let env, stmt' = check_assign env (t, SStructDeref(lval, n))
          (t, SStructDeref(rval, n)) stmts fail in (env, stmt'))
          (env, stmts) sdecl.members
      | (l, r) when l = r ->
        env, SAssign(lval, rval) :: stmts
      | _ ->
        raise fail in


    let rec lvalue need_lvalue env stmts = function
        Id s -> let k, t, s' = find_symbol_table env.scope s in
          if k = KindGlobal && env.cur_qualifier <> CpuOnly then
            raise (Failure ("access to global variable " ^
              s ^ " not allowed in non-CPU-only function"))
          else if k = KindConstGlobal && need_lvalue then
            raise (Failure ("const global " ^ s ^ " is not an lvalue"))
          else
            env, stmts, (t, SId(s'))
      | StructDeref(e, m) as d ->
          let env, stmts, e' = lvalue need_lvalue env stmts e in
          let typ = fst e' in
          env, stmts, ((match typ with
              Struct s ->
                let stype = StringMap.find s struct_decls in
                (try
                  fst (List.find (fun b -> snd b = m) stype.members)
                with Not_found ->
                  raise (Failure ("struct " ^ s ^ " does not contain member 
```

```
           " ^
          m ^ "␣in␣" ^ string_of_expr d)))
      | Pipeline p ->
        let ptype = StringMap.find p pipeline_decls in
        (try
         fst (List.find (fun b -> snd b = m) ptype.smembers)
        with Not_found ->
         raise (Failure ("pipeline␣" ^ p ^ "␣does␣not␣contain␣" ^
          m ^ "␣in␣" ^ string_of_expr d)))
      | Mat(b, 1, w) ->
        (match m with
          "x" | "y" when w >= 2 -> Mat(b, 1, 1)
         | "z" when w >= 3 -> Mat(b, 1, 1)
         | "w" when w = 4 -> Mat(b, 1, 1)
         | _ -> raise (Failure ("dereference␣of␣nonexistant␣member␣
           " ^ m ^
           "␣of␣a␣vector")))
      | Mat(b, w, l) ->
         (match m with
          "x" | "y" when w >= 2 -> Mat(b, 1, l)
         | "z" when w >= 3 -> Mat(b, 1, l)
         | "w" when w = 4 -> Mat(b, 1, l)
         | _ -> raise (Failure ("dereference␣of␣nonexistant␣member␣
           " ^ m ^
           "␣of␣a␣matrix")))
      | _ -> raise (Failure ("illegal␣dereference␣of␣type␣" ^
        string_of_typ typ ^ "␣in␣" ^ string_of_expr d)))
     , SStructDeref(e',␣m))
␣␣␣␣␣␣␣|␣ArrayDeref(e,␣i)␣as␣d␣->
␣␣␣␣␣␣␣␣␣␣let␣env,␣stmts,␣e' = lvalue need_lvalue env stmts e in
     let env, stmts, i'␣=␣expr␣env␣stmts␣i␣in
␣␣␣␣␣␣␣␣␣␣if␣fst␣i' <> Mat(Int, 1, 1) then
      raise (Failure ("index␣expression␣of␣of␣type␣" ^
       string_of_typ (fst i')␣^␣"␣instead␣of␣int␣in␣"␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣string_of_expr␣d))
␣␣␣␣␣␣␣␣␣␣else␣env,␣stmts,␣(match␣fst␣e' with
       Array(t, Some(_)) -> (t, SArrayDeref(e',␣i'))
      | Array(t, None) -> if env.cur_qualifier = CpuOnly
        then (t, SArrayDeref(e',␣i'))
        else raise (Failure "variable␣sized␣arrays␣cannot␣be␣used␣in␣
          GPU␣code")
      | _ -> raise (Failure ("array␣dereference␣of␣non-array␣type␣in␣
        " ^
           string_of_expr d)))
   | _ as e ->
     if need_lvalue then
      raise (Failure ("expression␣" ^ string_of_expr e ^ "␣is␣not␣an␣
        lvalue"))
```

```
        else
          expr env stmts e

(* Return the type of an expression and new expression or throw an
    exception *)
and expr (env : translation_environment) stmts = function
IntLit(l) -> env, stmts, (Mat(Int, 1, 1), SIntLit(l))
  | FloatLit(l) -> env, stmts, (Mat(Float, 1, 1), SFloatLit(l))
  | BoolLit(l) -> env, stmts, (Mat(Bool, 1, 1), SBoolLit(l))
  | CharLit(c) -> env, stmts, (Mat(Byte, 1, 1), SCharLit(c))
  | StringLit(s) ->
      env, stmts, (Array(Mat(Byte, 1, 1), Some (String.length s)),
          SStringLit(s))
  | Id _ | StructDeref(_, _) | ArrayDeref(_, _) as e ->
      lvalue false env stmts e
  | Binop(e1, op, e2) as e ->
     let env, stmts, e1 = expr env stmts e1 in
     let env, stmts, e2 = expr env stmts e2 in
     let t1 = fst e1 and t2 = fst e2 in
     let typ, op = (match op, t1, t2 with
       | Add,    Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Int, 1, l), IAdd)
       | Sub,    Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Int, 1, l), ISub)
       | Mult,   Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Int, 1, l), IMult)
       | Div,    Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Int, 1, l), IDiv)
       | Mod,   Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Int, 1, l), IMod)
       | Equal,  Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Bool, 1, 1), IEqual)
       | Neq,    Mat(Int, 1, l), Mat(Int, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Bool, 1, 1), INeq)
       | Add,    Mat(Float, w, l), Mat(Float, w',␣l') when w = w'␣&&␣l␣=␣
           l'
                 -> (Mat(Float, w, l), FAdd)
       | Sub,    Mat(Float, w, l), Mat(Float, w',␣l') when w = w'␣&&␣l␣=␣
           l'
                 -> (Mat(Float, w, l), FSub)
       | Mult,   Mat(Float, 1, l), Mat(Float, 1, l')␣␣when␣l␣=␣l'
                 -> (Mat(Float, 1, l), FMult)
       | Mult,   Mat(Float, w, l), Mat(Float, w',␣l') when w = l'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣->␣(Mat(Float,␣w', l), FMatMult)
       | Mult,   Mat(Float, 1, 1), Mat(Float, w, l)
                 -> (Mat(Float, w, l), Splat)
       | Div,    Mat(Float, 1, 1), Mat(Float, 1, l')␣when␣l␣=␣l'
                 -> (Mat(Float, 1, l), FDiv)
```

```
    | Equal,  Mat(Float, w, l), Mat(Float, w',␣l') when l = l'␣&&␣w␣=␣
      w'
            -> (Mat(Bool, 1, 1), FEqual)
    | Neq,    Mat(Float, w, l), Mat(Float, w',␣l') when l = l'␣&&␣w␣=␣
      w'
            -> (Mat(Bool, 1, 1), FNeq)
    | Less,   Mat(Float, w, l), Mat(Float, w',␣l') when l = l'␣&&␣w␣=␣
      w'
            -> (Mat(Bool, 1, 1), FLess)
    | Leq,    Mat(Float, w, l), Mat(Float, w',␣l') when l = l'␣&&␣w␣=␣
      w'
            -> (Mat(Bool, 1, 1), FLeq)
    | Greater, Mat(Float, w, l), Mat(Float, w',␣l') when l = l'␣&&␣w␣
      =␣w'
            -> (Mat(Bool, 1, 1), FGreater)
    | Geq,    Mat(Float, w, l), Mat(Float, w',␣l') when l = l'␣&&␣w␣=␣
      w'
            -> (Mat(Bool, 1, 1), FGeq)
    | Equal,  Mat(Bool, w, l), Mat(Bool, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), BEqual)
    | Neq,    Mat(Bool, w, l), Mat(Bool, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), BNeq)
    | Less,   Mat(Int, w, l), Mat(Int, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), ILess)
    | Leq,    Mat(Int, w, l), Mat(Int, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), ILeq)
    | Greater, Mat(Int, w, l), Mat(Int, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), IGreater)
    | Geq,    Mat(Int, w, l), Mat(Int, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), IGeq)
    | And,    Mat(Bool, w, l), Mat(Bool, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, w, l), BAnd)
    | Or,     Mat(Bool, w, l), Mat(Bool, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, w, l), BOr)
    | Equal,  Mat(Byte, w, l), Mat(Byte, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), U8Equal)
    | Neq,    Mat(Byte, w, l), Mat(Byte, w',␣l') when l = l'␣&&␣w␣=␣w'
            -> (Mat(Bool, 1, 1), U8Neq)
    | _ -> raise (Failure ("illegal␣binary␣operator␣" ^
        string_of_typ t1 ^ "␣" ^ string_of_op op ^ "␣" ^
        string_of_typ t2 ^ "␣in␣" ^ string_of_expr e))
    )
  in env, stmts, (typ, SBinop(e1, op, e2))
| Unop(op, e) as ex -> let env, stmts, e = expr env stmts e in
   let t = fst e in
   (match op, t with
 Neg, Mat(Int, w, l) -> env, stmts, (Mat(Int, w, l), SUnop(INeg, e))
| Neg, Mat(Float, w, l) -> env, stmts, (Mat(Float, w, l), SUnop(FNeg,
```

```
    e))
| Not, Mat(Bool, 1, 1) -> env, stmts, (Mat(Bool, 1, 1), SUnop(BNot, e))
    | PostInc, Mat(Int, 1, 1) ->
          let env, tmp = add_tmp env (Mat(Int, 1, 1)) in
          let stmts = SAssign(tmp, (Mat(Int, 1, 1), snd e)) :: stmts in
          let stmts =
            SAssign(e, (Mat(Int, 1, 1),
                  SBinop(e, IAdd, (Mat(Int, 1, 1), SIntLit(1)))))
            :: stmts in env, stmts, tmp
    | PostDec, Mat(Int, 1, 1) ->
          let env, tmp = add_tmp env (Mat(Int, 1, 1)) in
          let stmts = SAssign(tmp, (Mat(Int, 1, 1), snd e)) :: stmts in
          let stmts =
            SAssign(e, (Mat(Int, 1, 1),
                  SBinop(e, ISub, (Mat(Int, 1, 1), SIntLit(1)))))
            :: stmts in env, stmts, tmp
    | PreInc, Mat(Int, 1, 1) ->
          let stmts =
            SAssign(e, (Mat(Int, 1, 1),
                  SBinop(e, IAdd, (Mat(Int, 1, 1), SIntLit(1)))))
            :: stmts in env, stmts, e
    | PreDec, Mat(Int, 1, 1) ->
          let stmts =
            SAssign(e, (Mat(Int, 1, 1),
                  SBinop(e, ISub, (Mat(Int, 1, 1), SIntLit(1)))))
            :: stmts in env, stmts, e
    | _ -> raise (Failure ("illegal unary operator " ^ "in " ^
    string_of_expr ex)))
 | Noexpr -> env, stmts, (Void, SNoexpr)
 | Assign(lval, e) as ex ->
     let env, stmts, lval = lvalue true env stmts lval in
     let env, stmts, e = expr env stmts e in
     let env, stmts = check_assign env lval e stmts
       (Failure ("illegal assignment " ^ string_of_typ (fst lval) ^
         " = " ^ string_of_typ (fst e) ^ " in " ^ string_of_expr ex)) in
     env, stmts, lval
 | Call("length", [arr]) as call ->
     let env, stmts, arr = expr env stmts arr in
     let env, tmp = add_tmp env (Mat(Int, 1, 1)) in
     env, (match fst arr with
         Array(_, _) -> SCall(tmp, "length", [arr])
       | _ as typ ->
           raise (Failure ("expecting an array type instead of " ^
             string_of_typ typ ^ " in " ^ string_of_expr call))) ::
                 stmts,
     tmp
 | Call("upload_buffer", [buf; data]) as call ->
     check_call_qualifiers env "upload_buffer" CpuOnly;
```

```
     let env, stmts, buf = expr env stmts buf in
     let env, stmts, data = expr env stmts data in
     env, (match fst buf with
       Buffer(t) ->
        (match fst data with
          Array(t',␣_)␣->␣if␣t' = t then
           SCall((Void, SNoexpr), "upload_buffer", [buf; data])
          else
           raise (Failure ("buffer␣and␣array␣type␣do␣not␣match␣" ^
            "in␣" ^ string_of_expr call))
         | _ -> raise (Failure ("must␣upload␣an␣array␣in␣" ^
          "upload_buffer␣in␣" ^ string_of_expr call)))
      | _ -> raise (Failure ("first␣parameter␣to␣upload_buffer␣must␣
        be␣" ^
          "a␣buffer␣in␣" ^ string_of_expr call))) :: stmts,
     (Void, SNoexpr)
   | Call("draw", [p; i]) as call ->
     check_call_qualifiers env "draw" CpuOnly;
     let env, stmts, p'␣=␣expr␣env␣stmts␣p␣in
␣␣␣␣␣␣␣␣␣␣let␣env,␣stmts,␣i' = expr env stmts i in
     env, (match fst p',␣fst␣i' with
       Pipeline(_), Mat(Int, 1, 1) ->
        SCall((Void, SNoexpr), "draw", [p';␣i'])
      | _ -> raise (Failure ("invalid␣arguments␣to␣draw()␣in␣" ^
       string_of_expr call))) :: stmts,
     (Void, SNoexpr)
   | Call(fname, actuals) as call -> let fd = function_decl fname in
     check_call_qualifiers env fname fd.fqual;
     if List.length actuals != List.length fd.formals then
      raise (Failure ("expecting␣" ^ string_of_int
       (List.length fd.formals) ^ "␣arguments␣in␣" ^ string_of_expr
         call))
     else
      let env, stmts, actuals = List.fold_left2
       (* translate/evaluate function arguments *)
       (fun (env, stmts, actuals) (fq, _) e ->
        let env, stmts, se = if fq = In then
         expr env stmts e
        else
         lvalue true env stmts e in
        env, stmts, ((fq, se) :: actuals)) (env, stmts, [])
          fd.formals actuals in
      let actuals = List.rev actuals in
      (* make a temporary for each formal parameter *)
      let env, params = List.fold_left (fun (env, temps) (_, (ft, _))
         ->
       let env, temp = add_tmp env ft in
       (env, temp :: temps)) (env, []) fd.formals in
```

```
      let params = List.rev params in
      (* copy in-parameters to temporaries *)
      let env, stmts = List.fold_left2 (fun (env, stmts) temp (fq,
          actual) ->
        if fq = Out then
          env, stmts
        else
          let et = fst actual in
          let ft = fst temp in
          check_assign env temp actual stmts
            (Failure ("illegal␣actual␣argument␣found␣" ^ string_of_typ
                et ^
            "␣expected␣" ^ string_of_typ ft ^ "␣in␣" ^ string_of_expr
                call)))
      (env, stmts) params actuals in
      (* make call *)
      let env, ret_tmp = if fd.typ = Void then
        env, (Void, SNoexpr)
      else
        let env, tmp = add_tmp env fd.typ in
        env, tmp in
      let stmts = SCall(ret_tmp, fd.fname, params) :: stmts in
      (* copy temporaries to out-parameters *)
      let env, stmts = List.fold_left2 (fun (env, stmts) temp (fq,
          actual) ->
        if fq = In then
          env, stmts
        else
          let et = fst actual in
          let ft = fst temp in
          check_assign env actual temp stmts
            (Failure ("illegal␣actual␣argument␣found␣" ^ string_of_typ
                et ^
            "␣expected␣" ^ string_of_typ ft ^ "␣in␣" ^ string_of_expr
                call)))
       (env, stmts) params actuals in
      (* return the temporary we made for the call *)
      env, stmts, ret_tmp
| TypeCons(typ, actuals) as cons ->

    (* Take a list of pairs, with the formal arguments and the
     * corresponding *action* to be taken, which can insert
         statements,
     * modify the environment, and return a value. Try each possible
     * formal arguments, swallowing any errors and rolling back
         changes to
     * the environment, and only throw an error if all the
         possibilities
```

```
 * returned an error. This is similar to pattern-matching against
 * possible inputs, except the patterns to match against can be
 * generated at run-time for things like struct constructors and
 * user-defined functions. We can also more easily handle implicit
 * conversions, since we can just try to do the conversion and
   move to
 * the next pattern if it fails (thanks to the 100% purely
   functional
 * translation environment).
 *)
let check_cons action_list =
  let rec check_cons'␣=␣function
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(formals,␣builder)␣::␣rest␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣List.length␣actuals␣!=␣List.length␣formals␣then
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣check_cons' rest
                else (try
                  let env, stmts, actuals = List.fold_left
                    (* translate/evaluate function arguments *)
                    (fun (env, stmts, actuals) e ->
                      let env, stmts, se = expr env stmts e in
                      env, stmts, se :: actuals) (env, stmts, []) actuals in
                  let actuals = List.rev actuals in
                  (* make a temporary for each formal parameter *)
                  let env, params = List.fold_left (fun (env, temps) ft ->
                    let env, temp = add_tmp env ft in
                    (env, temp :: temps)) (env, []) formals in
                  let params = List.rev params in
                  (* copy in-parameters to temporaries *)
                  let env, stmts = List.fold_left2 (fun (env, stmts) temp
                      actual ->
                    check_assign env temp actual stmts
                      (Failure "dummy"))
                  (env, stmts) params actuals in
                  builder env stmts params
                with Failure(_) -> check_cons'␣rest)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣[]␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣raise␣(Failure␣("couldn't find matching formal
    arguments"␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" in "␣^␣string_of_expr␣cons␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", possible arguments are: \n"␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣String.concat␣"\n"␣(List.map␣(fun␣(formals,␣␣_)␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣String.concat␣", "␣(List.map␣(fun␣formal␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣string_of_typ␣formal)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣formals))
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣action_list)))
␣␣␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣␣␣␣␣␣␣check_cons'␣action_list
␣␣␣␣␣␣␣␣␣␣in
```

```
          (* action that simply returns a STypeCons with the parameters *)
          let action_cons env stmts params =
            env, stmts, (typ, STypeCons(params))
          in
          (* return a formal list that's "size" copies of "base_type".
            *)
          let array_vec_formals base_type size =
            let rec copies n =
                if n = 0 then [] else base_type :: copies (n-1) in
              (copies size)
          in

          (* action that creates a SUnop with the given op *)
          let action_unop op = (fun env stmts params ->
            env, stmts, (typ, SUnop(op, List.hd params)))
          in

          check_cons (match typ with
              | Struct s ->
                  let sdecl = try StringMap.find s struct_decls
                  with Not_found ->
                    raise (Failure ("struct " ^ s ^ " does not exist in " ^
                      string_of_expr cons))
                  in
                  let formals = List.map fst sdecl.members in
                  [formals, (fun env stmts params ->
                    let env, tmp = add_tmp env typ in
                    let stmts = List.fold_left2
                      (fun stmts (typ, name) actual ->
                        SAssign((typ, SStructDeref(tmp, name)), actual) ::
    stmts)
                    stmts sdecl.members params in
                    env, stmts, tmp)]
              | Mat(Float, 1, 1) ->
                [[Mat(Int, 1, 1)], action_unop Int2Float;
                 [Mat(Bool, 1, 1)], action_unop Bool2Float]
              | Mat(Int, 1, 1) ->
                [[Mat(Float, 1, 1)], action_unop Float2Int;
                 [Mat(Bool, 1, 1)], action_unop Bool2Int]
              | Mat(Int, 1, w) ->
                  [array_vec_formals (Mat(Int, 1, 1)) w, action_cons;
                    [Mat(Float, 1, w)], action_unop Float2Int;
                    [Mat(Bool, 1, w)], action_unop Bool2Int]
              | Mat(Float, 1, w) ->
                  [array_vec_formals (Mat(Float, 1, 1)) w, action_cons;
                    [Mat(Int, 1, w)], action_unop Int2Float;
                    [Mat(Bool, 1, w)], action_unop Bool2Float]
```

```
            | Mat(Bool, 1, w) ->
                [array_vec_formals (Mat(Bool, 1, 1)) w, action_cons;
                  [Mat(Float, 1, w)], action_unop Float2Bool;
                  [Mat(Int, 1, w)], action_unop Int2Bool]
            | Mat(b, 1, w) -> [array_vec_formals (Mat(b, 1, 1)) w, 
    action_cons]
            | Mat(b, w, 1) -> [array_vec_formals (Mat(b, 1, 1)) w, 
    action_cons]
            | Array(t, Some s) -> [array_vec_formals t s, action_cons]
            | Array(_, None) -> [[Mat(Int, 1, 1)], action_cons]
            | Buffer(t) -> check_buffer_type t; [[], action_cons]
            | Pipeline(_) -> [[Mat(Bool, 1, 1)], action_cons]
            | Window ->
                [[Mat(Int, 1, 1); Mat(Int, 1, 1); Mat(Bool, 1, 1)], 
    action_cons]
            | _ -> raise (Failure ("bad type constructor " ^
                      string_of_expr cons))

    in

    let check_bool_expr env stmts e =
      let env, stmts, se = expr env stmts e in
        if fst se <> Mat(Bool, 1, 1) then
          raise (Failure ("expected Boolean expression in " ^ 
    string_of_expr e))
        else env, stmts, se in

    let check_in_loop env = if env.in_loop then () else
      raise (Failure ("break/continue must be inside a loop")) in

    (* Verify a statement or throw an exception *)
    let rec check_stmt env in_loop_new = function
        Local((_, s), _) -> raise (Failure ("local variable " ^ s ^
          " not declared inside a block"))
      | _ as s ->
          let env', sl = stmts' { env with in_loop = in_loop_new; } [] [s] 
    in
          { env with locals = env'.locals; names = env'.names; }, List.rev 
    sl
    (* Helper function that returns the list of SAST statements in reverse
     * order *)
    and stmts' env sstmts sl = List.fold_left
      (fun (env, sstmts) stmt ->
        match sstmts with
            SBreak :: _ -> raise (Failure "nothing may follow a break")
          | SContinue :: _ -> raise (Failure "nothing may follow a 
    continue")
          | SReturn _ :: _ -> raise (Failure "nothing may follow a return")
```

```
          | _ -> match stmt with
              Break -> check_in_loop env; env, (SBreak :: sstmts)
            | Continue -> check_in_loop env; env, (SContinue :: sstmts)
            | Return e ->
                let env, sstmts, se = expr env sstmts e in
                let env, tmp = add_tmp env func.typ in
                let env, sstmts = check_assign env tmp se sstmts
                  (Failure ("return gives " ^ string_of_typ (fst se) ^ " expected " ^
                           string_of_typ func.typ ^ " in " ^ string_of_expr e))
                in
                env, (SReturn(tmp) :: sstmts)
            | Block sl -> let env', sstmts = stmts' env sstmts sl in
                { env with locals = env'.locals; names = env'.names; }, sstmts
            | If(p, b1, b2) ->
                let env, sstmts, p = check_bool_expr env sstmts p in
                let env, sthen = check_stmt env env.in_loop b1 in
                let env, selse = check_stmt env env.in_loop b2 in
                env, (SIf(p, sthen, selse) :: sstmts)
            | For(e1, e2, e3, st) ->
                let env, sstmts, _ = expr env sstmts e1 in
                let env, cond_stmts = check_stmt env true
                  (If (e2, Block([]), Break)) in (* if (!e2) break; *)
                let env, continue_stmts =
                  check_stmt env false (Expr(e3)) in
                let env, body = check_stmt env true st in
                env, (SLoop(cond_stmts @ body, continue_stmts) :: sstmts)
            | While(p, s) ->
                let env, cond_stmts = check_stmt env true
                  (If (p, Block([]), Break)) in (* if (!p) break; *)
                let env, body = check_stmt env true s in
                env, (SLoop(cond_stmts @ body, []) :: sstmts)
            | Expr e ->
                let env, sstmts, _ = expr env sstmts e in
                env, sstmts
            | Local ((t, s) as b, oe) ->
                (check_type
                  (fun s n -> s ^ " does not exist for local " ^ n ^
                    " in " ^ func.fname)
                  (fun n -> "illegal void local " ^ n ^
                                  " in " ^ func.fname) b);
                (* evaluate the initializer before we add to the symbol table to
                 * make sure the new name isn't available
                 *)
                let env, sstmts, e' =
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣match␣oe␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Some␣e␣->␣let␣env,␣sstmts,␣e'␣=␣expr␣env␣sstmts␣e␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣env,␣sstmts,␣Some␣(e',␣e)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣None␣->␣env,␣sstmts,␣None␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣env,␣name␣=␣add_symbol_table␣env␣s␣KindLocal␣t␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣env␣=␣{␣env␣with␣locals␣=␣(t,␣name)␣::␣env.locals␣}␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣match␣e'␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Some␣(e',␣e)␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣env,␣sstmts␣=␣check_assign␣env␣(t,␣SId␣name)␣e'␣
    sstmts
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(Failure␣("illegal initialization "␣^␣
    string_of_typ␣t␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" = "␣^␣string_of_typ␣(fst␣e')␣^␣" in "␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣string_of_typ␣t␣^␣" "␣^␣s␣^␣" = "␣^␣
    string_of_expr␣e␣^
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣";"))␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣env,␣sstmts
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣None␣->␣env,␣sstmts)
␣␣␣␣␣␣(env,␣sstmts)␣sl
␣␣␣␣in

␣␣␣␣(*␣check␣return␣type␣of␣shaders␣*)
␣␣␣␣(match␣func.fqual␣with
␣␣␣␣␣␣␣␣Vertex␣->␣if␣func.typ␣<>␣Mat(Float,␣1,␣4)␣then
␣␣␣␣␣␣␣␣␣␣raise␣(Failure␣("vertex entrypoint "␣^␣func.fname␣^
␣␣␣␣␣␣␣␣␣␣␣␣" must return vec4"))
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣()
␣␣␣␣␣␣|␣Fragment␣->␣if␣func.typ␣<>␣Void␣then
␣␣␣␣␣␣␣␣␣␣raise␣(Failure␣("fragment entrypoint "␣^␣func.fname␣^
␣␣␣␣␣␣␣␣␣␣␣␣" must return void"))
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣()
␣␣␣␣␣␣|␣_␣->␣())
␣␣␣␣;

␣␣␣␣let␣env␣=␣{␣env␣with␣cur_qualifier␣=␣func.fqual␣}␣in

␣␣␣␣let␣env,␣formals␣=␣List.fold_left␣(fun␣(env,␣formals)␣(q,␣(t,␣s))␣->
␣␣␣␣␣␣let␣env,␣name␣=␣add_symbol_table␣env␣s␣KindLocal␣t␣in
␣␣␣␣␣␣env,␣(q,␣(t,␣name))␣::␣formals)␣(env,␣[])␣func.formals
␣␣␣␣in

␣␣␣␣let␣formals␣=␣List.rev␣formals
␣␣␣␣in

␣␣␣␣let␣env,␣sbody␣=␣stmts'␣env␣[]␣func.body␣in
␣␣␣␣if␣func.typ␣<>␣Void␣then␣match␣sbody␣with
```

```
        SReturn _ :: _ -> ()
      | _ -> raise (Failure ("missing final return from function " ^ 
    func.fname ^
                    " with non-void return type"))
    else ()
    ;

    {
      styp = func.typ;
      sfname = func.fname;
      sfqual = func.fqual;
      sformals = formals;
      slocals = env.locals;
      sbody = List.rev sbody;
    }


  in

  let functions = List.map check_function functions
  in

  let function_decls = List.fold_left (fun m fd -> StringMap.add fd.sfname 
    fd m)
                         StringMap.empty functions
  in

  (* do a topological sort of the GPU-only function call graph to check for
   * loops, and to ensure that functions are always defined before they're
   * called for the GLSL backend since GLSL cares about the ordering.
   *)
  let func_succs fdecl =
    fold_sfdecl_pre (fun calls stmt ->
      match stmt with
          SCall(_, name, _) ->
            (try
              StringMap.find name function_decls :: calls
            (* since we already did semantic checking, we can ignore calls 
    to
             * functions that don't exist as they must be to built-in 
    functions
             *)
            with Not_found -> calls)
        | _ -> calls)
    [] fdecl
  in
  let gpu_functions = List.filter (fun fdecl ->
    match fdecl.sfqual with
```

```
␣␣␣␣␣␣␣␣GpuOnly␣|␣Fragment␣|␣Vertex␣|␣Both␣->␣true
␣␣␣␣␣␣|␣CpuOnly␣->␣false)␣functions
␣␣in
␣␣let␣cpu_functions␣=␣List.filter␣(fun␣fdecl␣->
␣␣␣␣fdecl.sfqual␣=␣CpuOnly)␣functions
␣␣in
␣␣let␣gpu_functions␣=␣List.rev␣(tsort␣gpu_functions␣func_succs␣(fun␣cycle␣
    ->
␣␣␣␣raise␣(Failure␣("recursive call by not-CPU-only functions: "␣^
␣␣␣␣␣␣String.concat␣" -> "␣(List.map␣(fun␣f␣->␣f.sfname)␣cycle))))
␣␣in
␣␣(structs,␣pipelines,␣globals,␣gpu_functions␣@␣cpu_functions)
```

parser.mly

```
/* Ocamlyacc parser for MicroC */

%{
open Ast
%}

%token SEMI
%token LPAREN RPAREN
%token LBRACE RBRACE
%token LBRACKET RBRACKET
%token COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT DOT INC DEC
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR MOD
%token RETURN BREAK CONTINUE IF ELSE FOR WHILE
%token CONST
%token VOID STRUCT PIPELINE BUFFER WINDOW
%token GPUONLY GPU VERTEX FRAGMENT
%token OUT INOUT UNIFORM
%token <int> BOOL
%token <int> BYTE
%token <int * int> FLOAT
%token <int> INT
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <char> CHAR_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
```

```
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT NEG PREINC PREDEC
%left DOT LPAREN LBRACKET POSTDEC POSTINC INC DEC

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
  /* nothing */ { { struct_decls = []; pipeline_decls = []; var_decls =
      []; func_decls = []; } }
 | decls vdecl { { $1 with var_decls = $2 :: $1.var_decls; } }
 | decls fdecl { { $1 with func_decls = $2 :: $1.func_decls; } }
 | decls sdecl { { $1 with struct_decls = $2 :: $1.struct_decls; } }
 | decls pdecl { { $1 with pipeline_decls = $2 :: $1.pipeline_decls; } }

fdecl:
  /* this definition has to be repeated to avoid a shift-reduce
      conflict... grr
   */
   func_qualifier typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { { typ = $2;
     fname = $3;
         fqual = $1;
     formals = $5;
     body = List.rev $8 } }
 | typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { { typ = $1;
     fname = $2;
         fqual = CpuOnly;
     formals = $4;
     body = List.rev $7 } }

func_qualifier:
    GPUONLY           { GpuOnly }
  | VERTEX            { Vertex }
  | FRAGMENT          { Fragment }
  | GPU               { Both }

vdecl:
```

```
    bind SEMI { (GVNone, $1, None) }
  | bind ASSIGN expr SEMI { (GVNone, $1, Some $3) }
  | CONST bind ASSIGN expr SEMI { (GVConst, $2, Some $4) }

simple_vdecl:
    bind SEMI { $1 }

simple_vdecl_list:
    /* nothing */ { [] }
  | simple_vdecl_list simple_vdecl { $2 :: $1 }

sdecl:
    STRUCT ID LBRACE simple_vdecl_list RBRACE SEMI { {
      sname = $2;
      members = List.rev $4;
    } }

pdecl:
    PIPELINE ID LBRACE pdecl_list RBRACE SEMI { {
      $4 with pname = $2;
    } }

pdecl_list:
  /* nothing */ { { pname = ""; fshader = ""; vshader = "" } }
  | pdecl_list VERTEX ID SEMI { if $1.vshader <> "" then
      raise (Failure ("vertex shader declared twice")) else
    { $1 with vshader = $3 } }
  | pdecl_list FRAGMENT ID SEMI { if $1.fshader <> "" then
      raise (Failure ("fragment shader declared twice")) else
    { $1 with fshader = $3 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list { List.rev $1 }

formal_list:
    formal_qualifier bind               { [($1,$2)] }
  | formal_list COMMA formal_qualifier bind { ($3,$4) :: $1 }

formal_qualifier:
    /* nothing */ { In }
  | OUT { Out }
  | INOUT { Inout }
  | UNIFORM { Uniform }

arrays:
    /* nothing */ { [] }
  | arrays LBRACKET INT_LITERAL RBRACKET { Some $3 :: $1 }
```

```
  | arrays LBRACKET RBRACKET { None :: $1 }

no_array_typ:
    INT { Mat(Int, 1, $1) }
  | FLOAT { Mat(Float, fst $1, snd $1) }
  | BOOL { Mat(Bool, 1, $1) }
  | BYTE { Mat(Byte, 1, $1) }
  | STRUCT ID { Struct($2) }
  | PIPELINE ID { Pipeline($2) }
  | BUFFER LT typ GT { Buffer($3) }
  | WINDOW { Window }
  | VOID { Void }

typ:
    no_array_typ arrays { List.fold_left (fun t len -> Array(t, len)) $1 $2
        }

bind:
  typ ID { ($1, $2) }

stmt_list:
    /* nothing */ { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr $1 }
  | bind SEMI { Local ($1, None) }
  | bind ASSIGN expr SEMI { Local ($1, Some $3) }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr        { $1 }

expr:
    INT_LITERAL    { IntLit($1) }
  | FLOAT_LITERAL  { FloatLit($1) }
  | TRUE           { BoolLit(true) }
  | FALSE          { BoolLit(false) }
```

```
  | CHAR_LITERAL   { CharLit($1) }
  | STRING_LITERAL { StringLit($1) }
  | ID             { Id($1) }
  | expr DOT   ID  { StructDeref($1, $3) }
  | expr LBRACKET expr RBRACKET { ArrayDeref($1, $3) }
  | expr PLUS  expr { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr TIMES expr { Binop($1, Mult, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr MOD expr { Binop($1, Mod, $3) }
  | expr EQ    expr { Binop($1, Equal, $3) }
  | expr NEQ   expr { Binop($1, Neq, $3) }
  | expr LT    expr { Binop($1, Less, $3) }
  | expr LEQ   expr { Binop($1, Leq, $3) }
  | expr GT    expr { Binop($1, Greater, $3) }
  | expr GEQ   expr { Binop($1, Geq, $3) }
  | expr AND   expr { Binop($1, And, $3) }
  | expr OR    expr { Binop($1, Or,  $3) }
  | expr INC %prec POSTINC { Unop(PostInc, $1) }
  | expr DEC %prec POSTDEC { Unop(PostDec, $1) }
  | INC expr %prec PREINC { Unop(PreInc, $2) }
  | DEC expr %prec PREDEC { Unop(PreDec, $2) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr       { Unop(Not, $2) }
  | expr ASSIGN expr { Assign($1, $3) }
  | typ LPAREN actuals_opt RPAREN { TypeCons($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

sast.ml

```
(* Lower-level Abstract Syntax Tree and functions for printing it *)

open Ast

type sop = IAdd | ISub | IMult | IDiv | IMod
        | IEqual | INeq | ILess | ILeq | IGreater | IGeq
        | FAdd | FSub | FMult | FDiv | FMatMult | Splat
        | FEqual | FNeq | FLess | FLeq | FGreater | FGeq
        | U8Equal | U8Neq
```

```
        | BAnd | BOr | BEqual | BNeq

type suop = INeg | FNeg | BNot |
           Int2Float | Float2Int | Bool2Int | Int2Bool | Bool2Float |
           Float2Bool

type sexpr_detail =
    SIntLit of int
  | SFloatLit of float
  | SBoolLit of bool
  | SCharLit of char
  | SStringLit of string
  | SId of string
  | SStructDeref of sexpr * string
  | SArrayDeref of sexpr * sexpr
  | SBinop of sexpr * sop * sexpr
  | SUnop of suop * sexpr
  | STypeCons of sexpr list
  | SNoexpr

and sexpr = typ * sexpr_detail

type sstmt =
    SAssign of sexpr * sexpr
  | SCall of sexpr * string * sexpr list
  | SReturn of sexpr
  | SIf of sexpr * sstmt list * sstmt list
  | SLoop of sstmt list * sstmt list (* body, continue statements *)
  | SBreak
  | SContinue

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sfqual : func_qualifier;
    sformals : (formal_qualifier * bind) list;
    slocals : bind list;
    sbody : sstmt list;
  }

type spipeline_decl = {
  spname : string;
  sfshader : string;
  svshader : string;
  sinputs : bind list;
  suniforms : bind list;
  smembers : bind list;
}
```

```
type svdecl = global_qualifier * bind * sexpr option

type sprogram = struct_decl list * spipeline_decl list * svdecl list *
    sfunc_decl list


(* do a pre-order traversal of all statements, calling 'f' and
 * accumulating the results
 *)
let fold_sfdecl_pre f a sfdecl =
  let rec fold_stmt_pre a stmt =
    let a = f a stmt in match stmt with
      | SIf(_, then_body, else_body) ->
          let a = fold_stmts_pre a then_body in
          fold_stmts_pre a else_body
      | SLoop(body, continue) ->
          let a = fold_stmts_pre a body in
          fold_stmts_pre a continue
      | SAssign(_, _) | SCall(_, _, _) | SReturn(_) | SBreak | SContinue ->
          a
  and fold_stmts_pre a elist =
    List.fold_left fold_stmt_pre a elist
  in

  fold_stmts_pre a sfdecl.sbody



(* Pretty-printing functions *)

let string_of_sop = function
    IAdd | FAdd -> "+"
  | ISub | FSub -> "-"
  | IMult | FMult | FMatMult | Splat -> "*"
  | IMod -> "%"
  | IDiv | FDiv -> "/"
  | IEqual | BEqual | FEqual | U8Equal -> "=="
  | INeq | BNeq | FNeq | U8Neq -> "!="
  | ILess | FLess -> "<"
  | ILeq | FLeq -> "<="
  | IGreater | FGreater -> ">"
  | IGeq | FGeq -> ">="
  | BAnd -> "&&"
  | BOr -> "||"

let string_of_suop = function
```

```
    INeg | FNeg -> "-"
  | BNot -> "!"
  | _ -> raise (Failure "shouldn't␣get␣here")

let rec string_of_sexpr (s : sexpr) = match snd s with
    SIntLit(l) -> string_of_int l
  | SFloatLit(l) -> string_of_float l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SCharLit(c) -> "'" ^ Char.escaped c ^ "'"
  | SStringLit(s) -> "\"" ^ String.escaped s ^ "\""
  | SId(s) -> s
  | SStructDeref(e, m) -> string_of_sexpr e ^ "." ^ m
  | SArrayDeref(e, i) -> string_of_sexpr e ^ "[" ^ string_of_sexpr i ^ "]"
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ "␣" ^ string_of_sop o ^ "␣" ^ string_of_sexpr e2
  | SUnop(Float2Int, e) -> "int(" ^ string_of_sexpr e ^ ")"
  | SUnop(Int2Float, e) -> "float(" ^ string_of_sexpr e ^ ")"
  | SUnop(Bool2Int, e) -> "int(" ^ string_of_sexpr e ^ ")"
  | SUnop(Bool2Float, e) -> "float(" ^ string_of_sexpr e ^ ")"
  | SUnop(o, e) -> string_of_suop o ^ string_of_sexpr e
  | STypeCons(el) ->
      string_of_typ (fst s) ^ "(" ^
      String.concat ",␣" (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> ""

let rec string_of_sstmt = function
    SAssign(v, e) -> string_of_sexpr v ^ "␣=␣" ^ string_of_sexpr e ^ ";\n"
  | SCall((Void, SNoexpr), f, el) ->
      f ^ "(" ^ String.concat ",␣" (List.map string_of_sexpr el) ^ ");\n"
  | SCall(ret, f, el) ->
      string_of_sexpr ret ^ "␣=␣" ^
      f ^ "(" ^ String.concat ",␣" (List.map string_of_sexpr el) ^ ");\n"
  | SReturn(expr) -> "return␣" ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, []) -> "if␣(" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmts
      s
  | SIf(e, s1, s2) -> "if␣(" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmts s1 ^ "else\n" ^ string_of_sstmts s2
  | SLoop(body, continue) ->
      "loop␣{\n" ^ String.concat "" (List.map string_of_sstmt body) ^
      "continue_block:\n" ^
      String.concat "" (List.map string_of_sstmt continue) ^ "}\n"
  | SBreak -> "break;\n"
  | SContinue -> "continue;\n"
and string_of_sstmts stmts =
  "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"

let string_of_sfdecl fdecl =
```

```
  string_of_func_qual fdecl.sfqual ^ "␣" ^ string_of_typ fdecl.styp ^ "␣" ^
  fdecl.sfname ^ "(" ^ String.concat ",␣" (List.map (fun (q, (t, n)) ->
  string_of_formal_qual q ^ "␣" ^ string_of_typ t ^ "␣" ^ n)
      fdecl.sformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_simple_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_spdecl pdecl =
  "pipeline␣" ^ pdecl.spname ^ "␣{\n" ^
  "@vertex␣" ^ pdecl.svshader ^ ";\n" ^
  "@fragment␣" ^ pdecl.sfshader ^ ";\n" ^
  String.concat ""
    (List.map (fun (t, n) -> "in␣" ^ string_of_typ t ^ "␣" ^ n ^ ";\n")
    pdecl.sinputs) ^
  "};\n"

let string_of_svdecl (qual, bind, init) = match init with
    None -> string_of_global_qual qual ^ string_of_bind bind ^ ";\n"
  | Some e -> string_of_global_qual qual ^ string_of_bind bind ^
      "␣=␣" ^ string_of_sexpr e ^ ";\n"

let string_of_sprogram (structs, pipelines, vars, funcs) =
  String.concat "" (List.map string_of_sdecl structs) ^ "\n" ^
  String.concat "" (List.map string_of_spdecl pipelines) ^ "\n" ^
  String.concat "" (List.map string_of_svdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

utils.ml

```
(* given a list of nodes, a function that returns the successors for a
    given
 * node, and a function to call when a cycle is detected, performs a
     topological
 * sort and returns the sorted list of nodes.
 *)
let tsort nodes succs cycle =
  let rec tsort'␣path␣visited␣=␣function
␣␣␣␣␣␣[]␣->␣visited
␣␣␣␣|␣n␣::␣nodes␣->
␣␣␣␣␣␣␣␣if␣List.mem␣n␣path␣then
␣␣␣␣␣␣␣␣␣␣(cycle␣(List.rev␣(n␣::␣path));␣visited)
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣let␣v' = if List.mem n visited then visited else
           n :: tsort'␣(n␣::␣path)␣visited␣(succs␣n)
␣␣␣␣␣␣␣␣␣␣in␣tsort' path v'␣nodes
␣␣in
```

```
  tsort' [] [] nodes
```

ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater |
    Geq |
          And | Or | Mod

type uop = Neg | Not | PreInc | PreDec | PostInc | PostDec

type base_type = Float | Int | Byte | Bool

type typ =
    Mat of base_type * int * int
  | Array of typ * int option
  | Struct of string
  | Buffer of typ
  | Pipeline of string
  | Window
  | Void

type bind = typ * string

type expr =
    IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | CharLit of char
  | StringLit of string
  | Id of string
  | StructDeref of expr * string
  | ArrayDeref of expr * expr
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | TypeCons of typ * expr list
  | Call of string * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Local of bind * expr option (* optional initializer *)
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
```

```
  | While of expr * stmt
  | Break
  | Continue

type formal_qualifier =
    In
  | Out
  | Inout
  | Uniform

type func_qualifier =
    GpuOnly
  | Vertex (* subset of GPU-only *)
  | Fragment (* subset of GPU-only *)
  | CpuOnly
  | Both

type func_decl = {
    typ : typ;
    fname : string;
    fqual : func_qualifier;
    formals : (formal_qualifier * bind) list;
    body : stmt list;
  }

type struct_decl = {
  sname : string;
  members : bind list;
}

type pipeline_decl = {
  pname : string;
  fshader : string;
  vshader : string;
}

type global_qualifier =
    GVNone
  | GVConst

type vdecl = global_qualifier * bind * expr option

type program = {
  struct_decls : struct_decl list;
  pipeline_decls : pipeline_decl list;
  var_decls : vdecl list;
  func_decls : func_decl list;
}
```

```
let rec base_type = function
    Array(typ, _) -> base_type typ
  | _ as typ -> typ

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Mod -> "%"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let rec string_of_typ = function
    Mat(Bool, 1, 1) -> "bool"
  | Mat(Int, 1, 1) -> "int"
  | Mat(Float, 1, 1) -> "float"
  | Mat(Byte, 1, 1) -> "u8"
  | Mat(Bool, 1, l) -> "bvec" ^ string_of_int l
  | Mat(Int, 1, l) -> "ivec" ^ string_of_int l
  | Mat(Float, 1, l) -> "vec" ^ string_of_int l
  | Mat(Byte, 1, l) -> "u8vec" ^ string_of_int l
  | Mat(Bool, w, l) -> "bmat" ^ string_of_int w ^ "x" ^ string_of_int l
  | Mat(Int, w, l) -> "imat" ^ string_of_int w ^ "x" ^ string_of_int l
  | Mat(Float, w, l) -> "mat" ^ string_of_int w ^ "x" ^ string_of_int l
  | Mat(Byte, w, l) -> "u8mat" ^ string_of_int w ^ "x" ^ string_of_int l
  | Struct s -> "struct " ^ s
  | Pipeline p -> "pipeline " ^ p
  | Buffer t -> "buffer" ^ "<" ^ string_of_typ t ^ ">"
  | Array(t, s) -> string_of_typ t ^ "[" ^
      (match s with Some(w) -> string_of_int w | _ -> "") ^ "]"
  | Window -> "window"
  | Void -> "void"

let rec string_of_expr expr =
  let string_of_uop o e = match o with
      Neg -> "-" ^ string_of_expr e
    | Not -> "!" ^ string_of_expr e
    | PreInc -> string_of_expr e ^ "++"
```

```
    | PostInc -> "++" ^ string_of_expr e
    | PreDec -> string_of_expr e ^ "--"
    | PostDec -> "--" ^ string_of_expr e
  in
  (match expr with
    IntLit(l) -> string_of_int l
  | FloatLit(l) -> string_of_float l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | CharLit(c) -> "'" ^ Char.escaped c ^ "'"
  | StringLit(s) -> "\"" ^ String.escaped s ^ "\""
  | Id(s) -> s
  | StructDeref(e, m) -> string_of_expr e ^ "." ^ m
  | ArrayDeref(e, i) -> string_of_expr e ^ "[" ^ string_of_expr i ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ "␣" ^ string_of_op o ^ "␣" ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o e
  | Assign(v, e) -> string_of_expr v ^ "␣=␣" ^ string_of_expr e
  | TypeCons(t, el) ->
      string_of_typ t ^ "(" ^ String.concat ",␣" (List.map string_of_expr
          el) ^ ")"
  | Call(fname, el) ->
      fname ^ "(" ^ String.concat ",␣" (List.map string_of_expr el) ^ ")"
  | Noexpr -> "")

let string_of_global_qual = function
    GVNone -> ""
  | GVConst -> "const␣"

let string_of_bind (t, id) =
  string_of_typ t ^ "␣" ^ id

let string_of_simple_vdecl bind = string_of_bind bind ^ ";\n"

let string_of_local_vdecl (bind, init) = match init with
    None -> string_of_bind bind ^ ";\n"
  | Some e -> string_of_bind bind ^
      "␣=␣" ^ string_of_expr e ^ ";\n"

let string_of_vdecl (qual, bind, init) =
  string_of_global_qual qual ^ string_of_local_vdecl (bind, init)

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Local(decl, e) -> string_of_local_vdecl (decl, e)
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return␣" ^ string_of_expr expr ^ ";\n";
```

```
  | If(e, s, Block([])) -> "if␣(" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s
  | If(e, s1, s2) -> "if␣(" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for␣(" ^ string_of_expr e1 ^ "␣;␣" ^ string_of_expr e2 ^ "␣;␣" ^
      string_of_expr e3 ^ ")␣" ^ string_of_stmt s
  | While(e, s) -> "while␣(" ^ string_of_expr e ^ ")␣" ^ string_of_stmt s
  | Break -> "break;"
  | Continue -> "continue;"

let string_of_formal_qual = function
    In -> ""
  | Out -> "out"
  | Inout -> "inout"
  | Uniform -> "uniform"

let string_of_func_qual = function
    CpuOnly -> "@cpuonly"
  | GpuOnly -> "@gpuonly"
  | Vertex -> "@vertex"
  | Fragment -> "@fragment"
  | Both -> "@gpu"

let string_of_fdecl fdecl =
  string_of_func_qual fdecl.fqual ^ "␣" ^ string_of_typ fdecl.typ ^ "␣" ^
  fdecl.fname ^ "(" ^ String.concat ",␣" (List.map (fun (q, (t, n)) ->
  string_of_formal_qual q ^ "␣" ^ string_of_typ t ^ "␣" ^ n)
      fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_sdecl sdecl =
  "struct␣" ^ sdecl.sname ^ "␣{\n" ^
  String.concat "" (List.map string_of_simple_vdecl sdecl.members) ^ "};\n"

let string_of_pdecl pdecl =
  "pipeline␣" ^ pdecl.pname ^ "␣{\n" ^
  "@vertex␣" ^ pdecl.vshader ^ ";\n" ^
  "@fragment␣" ^ pdecl.fshader ^ ";\n" ^
  "};\n"

let string_of_program prog =
  String.concat "" (List.map string_of_sdecl prog.struct_decls) ^ "\n" ^
  String.concat "" (List.map string_of_pdecl prog.pipeline_decls) ^ "\n" ^
  String.concat "" (List.map string_of_vdecl prog.var_decls) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl prog.func_decls)
```

codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

module L = Llvm
module A = Ast
module SA = Sast
module G = Glslcodegen

module StringMap = Map.Make(String)

(* helper function that returns the index of an element in a list
 * why isn't this a stdlib function?
 *)
let index_of e l =
  let rec index_of' i = function
      [] -> raise Not_found
    | hd :: tl -> if hd = e then i else index_of' (i + 1) tl
  in
index_of' 0 l

(* Why is this not a stdlib function? *)
let rec range i j = if i >= j then [] else i :: (range (i+1) j)

let translate ((structs, pipelines, globals, functions) as program :
    SA.sprogram) =
  let shaders = G.translate program in

  (* ignore GPU functions for the rest of the codegen *)
  let functions =
    List.filter (fun f -> f.SA.sfqual = A.CpuOnly || f.SA.sfqual = A.Both)
    functions in

  let context = L.global_context () in
```

```
let the_module = L.create_module context "MicroC"
and i32_t = L.i32_type context
and i8_t  = L.i8_type  context
and i1_t  = L.i1_type  context
and f32_t = L.float_type context
and f64_t = L.double_type context
and void_t = L.void_type context in
let string_t = L.pointer_type i8_t in
let voidp_t = L.pointer_type i8_t (* LLVM uses i8* instead of void* *) in

let make_vec_t base =
  [| base; L.array_type base 2;
          L.array_type base 3;
          L.array_type base 4 |]
in

let make_n_t base n =
    [| L.array_type base n;
       L.array_type (L.array_type base 2) n;
       L.array_type (L.array_type base 3) n;
       L.array_type (L.array_type base 4) n
    |]
in

let make_mat_t base =
    [| make_vec_t base; make_n_t base 2;
       make_n_t base 3; make_n_t base 4 |]
in

let vec_t = make_vec_t f32_t in
let ivec_t = make_vec_t i32_t in
let bvec_t = make_vec_t i1_t in
let byte_vec_t = make_vec_t i8_t in
let mat_t = make_mat_t f32_t in

let izero = L.const_int i32_t 0 in

(* define base pipeline type that every pipeline derives from
 * this is struct pipeline in runtime.c *)
let pipeline_t = L.struct_type context [|
  (* vertex_array *)
  i32_t;
  (* index_buffer *)
  i32_t;
  (* program *)
  i32_t;
  (* depth_func *)
  i32_t;
```

```
|] in

(* construct struct types *)
let struct_decls = List.fold_left (fun m s ->
  StringMap.add s.A.sname s m) StringMap.empty structs
in

let pipeline_decls = List.fold_left (fun m p ->
  StringMap.add p.SA.spname p m) StringMap.empty pipelines
in

let struct_types = List.fold_left (fun m s ->
  StringMap.add s.A.sname (L.named_struct_type context s.A.sname) m)
  StringMap.empty structs in

let rec ltype_of_typ = function
  | A.Mat(A.Int, 1, l) -> ivec_t.(l-1)
  | A.Mat(A.Bool, 1, l) -> bvec_t.(l-1)
  | A.Mat(A.Byte, 1, l) -> byte_vec_t.(l-1)
  | A.Mat(A.Float, w, l) -> mat_t.(w-1).(l-1)
  | A.Mat(_, _, _) -> raise (Failure "unimplemented")
  | A.Struct s -> StringMap.find s struct_types
  | A.Array(t, Some s) -> L.array_type (ltype_of_typ t) s
  | A.Array(t, None)-> L.struct_type context [| i32_t; L.pointer_type
      (ltype_of_typ t) |]
  | A.Window -> voidp_t
  | A.Pipeline(_) -> pipeline_t
  | A.Buffer(_) -> i32_t
  | A.Void -> void_t in

List.iter (fun s ->
  let llstruct = StringMap.find s.A.sname struct_types in
  L.struct_set_body llstruct
    (Array.of_list (List.map (fun m -> ltype_of_typ (fst m))
        s.A.members)) false)
structs;

let handle_const (_, detail) = match detail with
  SA.SIntLit i -> L.const_int i32_t i
    | SA.SFloatLit f -> L.const_float f32_t f
    | SA.SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
    | SA.SCharLit c -> L.const_int i8_t (Char.code c)
    | SA.SStringLit s -> L.const_string context s
    | _ -> raise (Failure "shouldn't get here")
in

(* Declare each global variable; remember its value in a map *)
let global_vars =
```

```
  let global_var m (_, (t, n), init) =
    let init = match init with
        Some e -> handle_const e
      | None -> L.undef (ltype_of_typ t)
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in

let shader_globals =
  StringMap.mapi (fun name shader ->
    L.define_global name (L.const_stringz context shader) the_module)
  shaders
in

let blis_string_t = ltype_of_typ (A.Array(A.Mat(A.Byte, 1, 1), None)) in

(* Declare printf(), which the print built-in function will call *)
let printf_t = L.var_arg_function_type i32_t [| voidp_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in

(* Declare functions in the built-in library that call into GLFW and
    OpenGL *)
let init_t = L.function_type void_t [| |] in
let init_func = L.declare_function "init" init_t the_module in
let create_window_t = L.function_type voidp_t [| i32_t; i32_t; i32_t |]
    in
let create_window_func =
  L.declare_function "create_window" create_window_t the_module in
let set_active_window_t = L.function_type void_t [| voidp_t |] in
let set_active_window_func =
  L.declare_function "set_active_window" set_active_window_t the_module in
let create_buffer_t = L.function_type i32_t [| |] in
let create_buffer_func = L.declare_function "create_buffer"
    create_buffer_t
  the_module in
let upload_buffer_t =
  L.function_type void_t [| i32_t; voidp_t; i32_t; i32_t |] in
let upload_buffer_func =
  L.declare_function "upload_buffer" upload_buffer_t the_module in
let create_pipeline_t =
  L.function_type void_t [|
    L.pointer_type pipeline_t; string_t; string_t; i32_t |] in
let create_pipeline_func =
  L.declare_function "create_pipeline" create_pipeline_t the_module in
let pipeline_bind_vertex_buffer_t = L.function_type void_t [|
  L.pointer_type pipeline_t; i32_t; i32_t; i32_t |] in
let pipeline_bind_vertex_buffer_func =
  L.declare_function "pipeline_bind_vertex_buffer"
    pipeline_bind_vertex_buffer_t the_module in
```

```
let pipeline_get_vertex_buffer_t = L.function_type i32_t [|
  L.pointer_type pipeline_t; i32_t |] in
let pipeline_get_vertex_buffer_func =
  L.declare_function "pipeline_get_vertex_buffer"
    pipeline_get_vertex_buffer_t the_module in
let pipeline_get_uniform_location_t =
  L.function_type i32_t [| L.pointer_type pipeline_t; string_t |] in
let pipeline_get_uniform_location_func =
  L.declare_function "pipeline_get_uniform_location"
    pipeline_get_uniform_location_t the_module in
let pipeline_set_uniform_float_t =
  L.function_type void_t
    [| L.pointer_type pipeline_t; i32_t; L.pointer_type f32_t; i32_t;
        i32_t |] in
let pipeline_set_uniform_float_func =
  L.declare_function "pipeline_set_uniform_float"
    pipeline_set_uniform_float_t the_module in
let pipeline_set_uniform_int_t =
  L.function_type void_t
    [| L.pointer_type pipeline_t; i32_t; L.pointer_type i32_t; i32_t;
        i32_t |] in
let pipeline_set_uniform_int_func =
  L.declare_function "pipeline_set_uniform_int"
    pipeline_set_uniform_int_t the_module in
let pipeline_get_uniform_float_t =
  L.function_type void_t
    [| L.pointer_type pipeline_t; i32_t; L.pointer_type f32_t |] in
let pipeline_get_uniform_float_func =
  L.declare_function "pipeline_get_uniform_float"
    pipeline_get_uniform_float_t the_module in
let pipeline_get_uniform_int_t =
  L.function_type void_t
    [| L.pointer_type pipeline_t; i32_t; L.pointer_type i32_t |] in
let pipeline_get_uniform_int_func =
  L.declare_function "pipeline_get_uniform_int"
    pipeline_get_uniform_int_t the_module in
let draw_arrays_t = L.function_type void_t [| L.pointer_type pipeline_t;
    i32_t |] in
let draw_arrays_func =
  L.declare_function "draw_arrays" draw_arrays_t the_module in
let clear_t =
  L.function_type void_t
    [| L.pointer_type (L.array_type f32_t 4) |] in
let clear_func =
  L.declare_function "clear" clear_t the_module in
let swap_buffers_t = L.function_type void_t [| voidp_t |] in
let swap_buffers_func =
  L.declare_function "glfwSwapBuffers" swap_buffers_t the_module in
```

```
let poll_events_t = L.function_type void_t [| |] in
let poll_events_func =
  L.declare_function "glfwPollEvents" poll_events_t the_module in
let get_key_t = L.function_type i32_t [| voidp_t; i32_t |] in
let get_key_func =
  L.declare_function "glfwGetKey" get_key_t the_module in
let get_mouse_t = L.function_type i32_t [| voidp_t; i32_t |] in
let get_mouse_func =
  L.declare_function "glfwGetMouseButton" get_mouse_t the_module in
let get_mouse_pos_t =
  L.function_type void_t
    [| voidp_t; L.pointer_type f64_t; L.pointer_type f64_t; |] in
let get_mouse_pos_func =
  L.declare_function "glfwGetCursorPos" get_mouse_pos_t the_module in
let should_close_t = L.function_type i32_t [| voidp_t |] in
let should_close_func =
  L.declare_function "glfwWindowShouldClose" should_close_t the_module in
let read_pixel_t =
  L.function_type void_t [| i32_t; i32_t; L.pointer_type vec_t.(3) |] in
let read_pixel_func =
  L.declare_function "read_pixel" read_pixel_t the_module in
let read_file_t =
  L.function_type void_t [| L.pointer_type blis_string_t; blis_string_t
      |] in
let read_file_func =
  L.declare_function "read_file" read_file_t the_module in
let print_string_t =
  L.function_type void_t [| blis_string_t |] in
let print_string_func =
  L.declare_function "print_string" print_string_t the_module in
let sin_t =
  L.function_type f32_t [| f32_t |]
in
let sin_func =
  L.declare_function "sinf" sin_t the_module
in
let cos_t =
  L.function_type f32_t [| f32_t |]
in
let cos_func =
  L.declare_function "cosf" cos_t the_module
in
let pow_t =
  L.function_type f32_t [| f32_t; f32_t |]
in
let pow_func =
  L.declare_function "powf" pow_t the_module
in
```

```
  let sqrt_t =
    L.function_type f32_t [| f32_t |]
  in
  let sqrt_func =
    L.declare_function "sqrtf" sqrt_t the_module
  in
  let floor_t =
    L.function_type f32_t [| f32_t |]
  in
  let floor_func =
    L.declare_function "floorf" floor_t the_module
  in

  (* Define each function (arguments and return type) so we can call it *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.SA.sfname
      and formal_types =
    Array.of_list (List.map (fun (q, (t,_)) ->
          let t'␣=␣ltype_of_typ␣t␣in
␣␣␣␣␣␣␣␣␣␣if␣q␣=␣A.In␣then␣t' else L.pointer_type t')␣fdecl.SA.sformals)
␣␣␣␣␣␣in␣let␣ftype␣=␣L.function_type␣(ltype_of_typ␣fdecl.SA.styp)␣
    formal_types␣in
␣␣␣␣␣␣StringMap.add␣name␣(L.define_function␣name␣ftype␣the_module,␣fdecl)␣
    m␣in
␣␣␣␣List.fold_left␣function_decl␣StringMap.empty␣functions␣in

␣␣(*␣Fill␣in␣the␣body␣of␣the␣given␣function␣*)
␣␣let␣build_function_body␣fdecl␣=
␣␣␣␣let␣(the_function,␣_)␣=␣StringMap.find␣fdecl.SA.sfname␣function_decls␣
    in
␣␣␣␣let␣builder␣=␣L.builder_at_end␣context␣(L.entry_block␣the_function)␣in

␣␣␣␣let␣int_format_str␣=␣L.build_global_stringptr␣"%d\n"␣"fmt"␣builder␣in
␣␣␣␣let␣float_format_str␣=␣L.build_global_stringptr␣"%f\n"␣"fmt"␣builder␣in
␣␣␣␣let␣char_format_str␣=␣L.build_global_stringptr␣"%c\n"␣"fmt"␣builder␣in

␣␣␣␣let␣add_formal␣m␣(q,␣(t,␣n))␣p␣=␣L.set_value_name␣n␣p;
␣␣␣␣␣␣let␣local␣=␣L.build_alloca␣(ltype_of_typ␣t)␣n␣builder␣in
␣␣␣␣␣␣(match␣q␣with
␣␣␣␣␣␣␣␣␣␣A.In␣->␣ignore␣(L.build_store␣p␣local␣builder)
␣␣␣␣␣␣␣␣|␣A.Inout␣->␣ignore␣(L.build_store
␣␣␣␣␣␣␣␣␣␣␣␣(L.build_load␣p␣"tmp"␣builder)␣local␣builder)
␣␣␣␣␣␣␣␣|␣A.Out␣->␣()
␣␣␣␣␣␣␣␣|␣A.Uniform␣->␣raise␣(Failure␣"unreachable"));
␣␣␣␣␣␣StringMap.add␣n␣local␣m␣in

␣␣␣␣let␣formals␣=␣List.fold_left2␣add_formal␣StringMap.empty␣
```

```
    fdecl.SA.sformals
␣␣␣␣␣␣␣␣(Array.to_list␣(L.params␣the_function))␣in

␣␣␣␣(*␣Construct␣the␣function’s "locals": formal arguments and locally
      declared variables. Allocate each on the stack, initialize their
      value, if appropriate, and remember their values in the "locals" map
          *)
    let local_vars =
      let add_local m (t, n) =
    let local_var = L.build_alloca (ltype_of_typ t) n builder
    in StringMap.add n local_var m in
      List.fold_left add_local formals fdecl.SA.slocals in

    (* Return the value for a variable or formal argument *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars
    in

    (* Given a pointer to matrix of a given type, return a pointer to the
        first
     * element.
     *)
    let mat_first_elem ptr cols rows builder =
      if cols = 1 && rows = 1 then
        ptr
      else if cols = 1 || rows = 1 then
        L.build_gep ptr [| izero; izero |] "" builder
      else
        L.build_gep ptr [| izero; izero; izero |] "" builder
    in

    (* Given an LLVM array type, explode it into an Ocaml list containing
        the
     * elements of the array.
     *)
    let explode_array arr name builder =
      let len = L.array_length (L.type_of arr) in
      List.rev (List.fold_left (fun lst idx ->
        (L.build_extractvalue arr idx name builder) :: lst) [] (range 0
            len))
    in

    (* Given a list of LLVM values, construct an LLVM array value containing
     * them.
     *)
    let make_array lst name builder =
      let len = List.length lst in
      let base_typ = L.type_of (List.hd lst) in
```

```
      fst (List.fold_left (fun (arr, idx) elem ->
        L.build_insertvalue arr elem idx name builder, idx + 1)
      (L.undef (L.array_type base_typ len), 0) lst)
    in

    (* apply f elem to each element of the LLVM array *)
    let map_array f arr name builder =
      make_array (List.map f (explode_array arr name builder)) name builder
    in

    (* apply f elem1 elem2 to each element of the LLVM arrays *)
    let map_array2 f arr1 arr2 name builder =
      let lst1 = explode_array arr1 name builder in
      let lst2 = explode_array arr2 name builder in
      make_array (List.map2 f lst1 lst2) name builder
    in

    let twod_array_wrap cols rows llval name builder =
      let llval = if cols != 1 then llval else
        make_array [llval] name builder
      in
      if rows != 1 then llval else
        map_array (fun elem ->
          make_array [elem] name builder) llval name builder
    in
    let twod_array_unwrap cols rows llval name builder =
      let llval = if rows != 1 then llval else
        map_array (fun elem ->
          List.hd (explode_array elem name builder)) llval name builder
      in
      if cols != 1 then llval else
        List.hd (explode_array llval name builder)
    in

    (* call f elem for each element of a Blis matrix type *)
    let map_blis_array f cols rows llval name builder =
      let llval = twod_array_wrap cols rows llval name builder
      in
      let llval'␣=␣map_array␣(fun␣elem␣->
␣␣␣␣␣␣␣␣map_array␣f␣elem␣name␣builder)␣llval␣name␣builder
␣␣␣␣␣␣in
␣␣␣␣␣␣twod_array_unwrap␣cols␣rows␣llval' name builder
    in

    (* call f elem1 elem2 for each element of the Blis matrix types *)
    let map_blis_array2 f cols rows llval1 llval2 name builder =
      let llval1 = twod_array_wrap cols rows llval1 name builder
      in
```

```
      let llval2 = twod_array_wrap cols rows llval2 name builder
      in
      let llval'␣=␣map_array2␣(fun␣elem1␣elem2␣->
␣␣␣␣␣␣␣␣map_array2␣f␣elem1␣elem2␣name␣builder)␣llval1␣llval2␣name␣builder
␣␣␣␣␣␣in
␣␣␣␣␣␣twod_array_unwrap␣cols␣rows␣llval' name builder
    in

    (* evaluates an expression and returns a pointer to its value. If the
     * expression is an lvalue, guarantees that the pointer is to the memory
     * referenced by the lvalue.
     *)
    let rec lvalue builder sexpr = match snd sexpr with
        SA.SId s -> lookup s
      | SA.SStructDeref (e, m) ->
          let e'␣=␣lvalue␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣(match␣fst␣e␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣A.Struct␣s␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣decl␣=␣StringMap.find␣s␣struct_decls␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣L.build_struct_gep␣e'
                (index_of m (List.map snd decl.A.members))
                "tmp" builder
            | A.Mat (_, _, _) ->
                L.build_gep e'␣[|izero;␣L.const_int␣i32_t␣(match␣m␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"x"␣->␣0
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣"y"␣->␣1
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣"z"␣->␣2
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣"w"␣->␣3
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣_␣->␣raise␣(Failure␣"shouldn't get here"))|]
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"tmp"␣builder
␣␣␣␣␣␣␣␣␣␣␣␣|␣A.Pipeline(_)␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣m␣=␣"indices"␣then
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣L.build_struct_gep␣e'␣1␣"tmp"␣builder
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣we␣can␣only␣get␣here␣when␣getting␣uniform/input␣
    variables,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣since␣handle_assign()␣takes␣care␣of␣the␣other␣cases.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣tmp␣=␣L.build_alloca␣(ltype_of_typ␣(fst␣sexpr))␣""␣
    builder
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_store␣(expr␣builder␣sexpr)␣tmp␣builder);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣tmp
␣␣␣␣␣␣␣␣␣␣␣␣|␣_␣->␣raise␣(Failure␣"unexpected type"))
␣␣␣␣␣␣|␣SA.SArrayDeref␣(e,␣i)␣->
␣␣␣␣␣␣␣␣␣␣let␣e'␣=␣lvalue␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣let␣i'␣=␣expr␣builder␣i␣in
␣␣␣␣␣␣␣␣␣␣(match␣(fst␣e)␣with
```

```
            A.Array(_, Some _) -> L.build_gep e' [| izero; i' |]
              "tmp" builder
          | A.Array(_, None) -> L.build_gep
            (L.build_extractvalue (L.build_load e' "" builder) 1 ""
    builder)
            [| i' |] "tmp" builder
          | _ -> raise (Failure "not supported"))
      | _ -> let e' = expr builder sexpr in
          let temp =
            L.build_alloca (ltype_of_typ (fst sexpr)) "expr_tmp" builder in
          ignore (L.build_store e' temp builder); temp

    and handle_assign builder l r =
      match l with
          (A.Buffer(A.Mat(A.Float, 1, comp)),
    SA.SStructDeref((A.Pipeline(p), _) as e, m)) ->
            let pdecl = StringMap.find p pipeline_decls in
            let location = index_of m (List.map snd pdecl.SA.sinputs) in
            let lval' = lvalue builder e in
            let e' = expr builder r in
            ignore (L.build_call pipeline_bind_vertex_buffer_func [|
              lval'; e'; L.const_int i32_t comp; L.const_int i32_t
    location |]
              "" builder)
        | (A.Mat(b, c, n), SA.SStructDeref((A.Pipeline(_), _) as e, m)) ->
            let lval' = lvalue builder e in
            let loc = L.build_call pipeline_get_uniform_location_func [|
              lval'; L.build_global_stringptr m "" builder |] "" builder in
            ignore (match b with
                A.Float ->
                  let e' = lvalue builder r in
                  L.build_call pipeline_set_uniform_float_func [|
                    lval'; loc; mat_first_elem e' c n builder;
                    L.const_int i32_t n; L.const_int i32_t c |] "" builder
              | A.Int ->
                  let e' = lvalue builder r in
                  L.build_call pipeline_set_uniform_int_func [|
                    lval'; loc; mat_first_elem e' c n builder;
                    L.const_int i32_t n; L.const_int i32_t c |] "" builder
              | A.Bool ->
                  let e' = lvalue builder (A.Mat(A.Int, c, n),
    SA.SUnop(SA.Bool2Int, r))
                  in
                  L.build_call pipeline_set_uniform_int_func [|
                    lval'; loc; mat_first_elem e' c n builder;
                    L.const_int i32_t n; L.const_int i32_t c |] "" builder
              | _ -> raise (Failure "unimplemented"));
        | _ -> let lval' = lvalue builder l in
```

```
            let e' = expr builder r in
            ignore (L.build_store e' lval' builder)

    (* Construct code for an expression; return its value *)
    and expr builder sexpr = match snd sexpr with
    SA.SIntLit _ | SA.SFloatLit _ | SA.SBoolLit _
      | SA.SCharLit _ | SA.SStringLit _ ->
          handle_const sexpr
      | SA.SNoexpr -> izero
      | SA.SStructDeref((A.Pipeline(_), _) as e, "indices") ->
          let e' = expr builder e in
          L.build_extractvalue e' 1 "" builder
      | SA.SStructDeref((A.Pipeline(p), _) as e, m) ->
          let pdecl = StringMap.find p pipeline_decls in
          let e' = lvalue builder e in
          (try
            let location = index_of m (List.map snd pdecl.SA.sinputs) in
            L.build_call pipeline_get_vertex_buffer_func [|
              e'; L.const_int i32_t location |] "" builder
          with Not_found ->
            let loc = L.build_call pipeline_get_uniform_location_func [|
              e'; L.build_global_stringptr m "" builder |] "" builder in
            (match fst sexpr with
                A.Mat(A.Float, c, r) ->
                  let tmp = L.build_alloca (ltype_of_typ (fst sexpr)) "" builder in
                  ignore (L.build_call pipeline_get_uniform_float_func [|
                    e'; loc; mat_first_elem tmp c r builder |]
                  "" builder);
                  L.build_load tmp "" builder
              | A.Mat(A.Int, c, r) ->
                  let tmp = L.build_alloca (ltype_of_typ (fst sexpr)) "" builder in
                  ignore (L.build_call pipeline_get_uniform_int_func [|
                    e'; loc; mat_first_elem tmp c r builder |]
                  "" builder);
                  L.build_load tmp "" builder
              | A.Mat(A.Bool, c, r) ->
                  let tmp = L.build_alloca
                    (ltype_of_typ (A.Mat(A.Int, c, r))) "" builder in

                  ignore (L.build_call pipeline_get_uniform_int_func [|
                    e'; loc; mat_first_elem tmp c r builder |]
                  "" builder);
                  let tmp = L.build_load tmp "" builder in
                  map_blis_array (fun e ->
                    L.build_icmp L.Icmp.Ne e izero "" builder) c r tmp ""
                    builder
```

```
              | _ -> raise (Failure "unimplemented")))
      | SA.SId _ | SA.SStructDeref (_, _) | SA.SArrayDeref (_, _) ->
          L.build_load (lvalue builder sexpr) "load_tmp" builder
      | SA.SBinop (e1, op, e2) ->
          let e1' = expr builder e1
      and e2' = expr builder e2 in
          let e1cols, e1rows, e2cols, e2rows = match fst e1, fst e2 with
                | A.Mat(_, w, l), A.Mat(_, w', l') ->
                    w, l, w', l'
                | _ -> raise (Failure "shouldn't get here");
          in
          let llvbase_type, cols, rows = match fst sexpr with
                | A.Mat(b, w, l) -> (ltype_of_typ (A.Mat(b, 1, 1))), w, l
                | _ -> raise (Failure "shouldn't get here");
          in
          let twod_array cols rows llvbase_type =
            L.undef (L.array_type (L.array_type llvbase_type rows) cols)
          in
          let per_component_builder op e1'' e2'' str builder =
            map_blis_array2 (fun elem1 elem2 -> op elem1 elem2 str builder)
              cols rows e1'' e2'' str builder
          in
          let dot_prod vec1 vec2 str builder =
              let val1 = L.build_extractvalue vec1 0 str builder in
              let val2 = L.build_extractvalue vec2 0 str builder in
              let val3 = L.build_fmul val1 val2 str builder in
            List.fold_left (fun sum index ->
              let val1' = L.build_extractvalue vec1 index str builder in
              let val2' = L.build_extractvalue vec2 index str builder in
              let val3' = L.build_fmul val1' val2' str builder in
              L.build_fadd val3' sum str builder) val3 (range 1 e1cols)
          in
          let mat_row_extract mat row str builder =
            List.fold_left (fun acc index ->
              let colm = L.build_extractvalue mat index str builder in
              let val1 = L.build_extractvalue colm row str builder in
              L.build_insertvalue acc val1 index "matrow" builder)
            (L.undef  (L.array_type llvbase_type e1cols)) (range 0 e1cols)
          in
          let mat_mult_col mat colvec str builder =
             List.fold_left (fun acc row ->
               let rowvec = mat_row_extract mat row str builder in
               let value = dot_prod rowvec colvec str builder in
               L.build_insertvalue acc value row "matcol" builder)
             (L.undef (L.array_type llvbase_type rows)) (range 0 rows)
          in
          let mat_mat_mult mat1 mat2 str builder =
             List.fold_left (fun acc col ->
```

```
               let x = L.build_extractvalue mat2 col str builder in
               let b = mat_mult_col mat1 x str builder in
              L.build_insertvalue acc b col "matmat" builder)
             (twod_array cols rows llvbase_type) (range 0 cols)
          in
          let fmat_mult mat1 mat2 str builder =
            let mat1 =
              twod_array_wrap e1cols e1rows mat1 str builder
            in
            let mat2 =
              twod_array_wrap e2cols e2rows mat2 str builder
            in
            let output = mat_mat_mult mat1 mat2 str builder in
            twod_array_unwrap cols rows output str builder
          in
          let scalar_expander value cols rows str builder =
            if cols = 1 && rows = 1 then value
            else if cols = 1 && rows != 1 then
              List.fold_left (fun acc row ->
                L.build_insertvalue acc value row str builder)
              (L.undef (L.array_type llvbase_type rows)) (range 0 rows)
            else if cols != 1 && rows = 1 then
              List.fold_left (fun acc col ->
                L.build_insertvalue acc value col str builder)
              (L.undef (L.array_type llvbase_type cols)) (range 0 cols)
            else
              let column = List.fold_left (fun acc row ->
                L.build_insertvalue acc value row str builder)
              (L.undef (L.array_type llvbase_type rows)) (range 0 rows)
              in
              List.fold_left (fun acc col ->
                L.build_insertvalue acc column col str builder)
              (twod_array cols rows llvbase_type) (range 0 cols)
          in
          let splat_mult e1 e2 str builder =
            per_component_builder L.build_fmul
              (scalar_expander e1 cols rows str builder) e2 str builder
          in
          let vec_vec_comparator op1 op2 vec1 vec2 str builder =
            let truth_start = L.const_int i1_t 1 in
            List.fold_left (fun acc row ->
               let val1 = L.build_extractvalue vec1 row str builder in
               let val2 = L.build_extractvalue vec2 row str builder in
               let comp12 = op1 op2 val1 val2 "vecl" builder in
               L.build_and acc comp12 "vecb" builder) truth_start (range 0
    e1rows)
          in
          let mat_mat_comparator op1 op2 mat1 mat2 str builder =
```

```
            let truth_start = L.const_int i1_t 1 in
            List.fold_left (fun acc col ->
               let vec1 = L.build_extractvalue mat1 col str builder in
               let vec2 = L.build_extractvalue mat2 col str builder in
               let comp12 = vec_vec_comparator op1 op2 vec1 vec2 "vec" 
    builder in
               L.build_and acc comp12 "mat" builder) truth_start (range 0 
    e1cols)
          in
          let component_comparator op1 op2 e1 e2 str builder =
              let mat1 =
                twod_array_wrap e1cols e1rows e1 str builder
              in
              let mat2 =
                twod_array_wrap e2cols e2rows e2 str builder
              in
              mat_mat_comparator op1 op2 mat1 mat2 str builder
          in
      (match op with
        SA.IAdd     -> per_component_builder L.build_add
      | SA.ISub     -> per_component_builder L.build_sub
      | SA.IMult    -> per_component_builder L.build_mul
          | SA.IMod     -> per_component_builder L.build_srem
          | SA.IDiv     -> per_component_builder L.build_sdiv
      | SA.IEqual   -> component_comparator L.build_icmp L.Icmp.Eq
      | SA.INeq     -> component_comparator L.build_icmp L.Icmp.Ne
      | SA.ILess    -> component_comparator L.build_icmp L.Icmp.Slt
      | SA.ILeq     -> component_comparator L.build_icmp L.Icmp.Sle
      | SA.IGreater -> component_comparator L.build_icmp L.Icmp.Sgt
      | SA.IGeq     -> component_comparator L.build_icmp L.Icmp.Sge
          | SA.FAdd     -> per_component_builder L.build_fadd
          | SA.FSub     -> per_component_builder L.build_fsub
          | SA.FMult    -> per_component_builder L.build_fmul
          | SA.FDiv     -> per_component_builder L.build_fdiv
          | SA.FMatMult -> fmat_mult
          | SA.Splat    -> splat_mult
      | SA.FEqual   -> component_comparator L.build_fcmp L.Fcmp.Oeq
      | SA.FNeq     -> component_comparator L.build_fcmp L.Fcmp.One
      | SA.FLess    -> component_comparator L.build_fcmp L.Fcmp.Olt
      | SA.FLeq     -> component_comparator L.build_fcmp L.Fcmp.Ole
      | SA.FGreater -> component_comparator L.build_fcmp L.Fcmp.Ogt
      | SA.FGeq     -> component_comparator L.build_fcmp L.Fcmp.Oge
          | SA.U8Equal  -> component_comparator L.build_icmp L.Icmp.Eq
          | SA.U8Neq    -> component_comparator L.build_icmp L.Icmp.Ne
      | SA.BAnd     -> per_component_builder L.build_and
      | SA.BOr      -> per_component_builder L.build_or
      | SA.BEqual   -> component_comparator L.build_icmp L.Icmp.Eq
      | SA.BNeq     -> component_comparator L.build_icmp L.Icmp.Ne
```

```
␣␣␣␣␣␣)␣e1'␣e2'␣"tmp"␣builder
␣␣␣␣␣␣|␣SA.SUnop(op,␣e)␣->
␣␣␣␣␣␣let␣e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣let␣cols,␣rows␣=␣match␣fst␣sexpr␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣A.Mat(_,␣w,␣l)␣->␣w,␣l
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣_␣->␣raise␣(Failure␣"shouldn't␣get␣here");
␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣␣␣␣␣let␣per_component_builder␣op␣e''␣str␣builder␣=
␣␣␣␣␣␣␣␣␣␣␣␣map_blis_array␣(fun␣elem␣->␣op␣elem␣str␣builder)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣cols␣rows␣e''␣str␣builder
␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣(match␣op␣with
␣␣␣␣␣␣␣␣SA.INeg␣␣␣␣␣␣␣->␣per_component_builder␣L.build_neg␣e'
      | SA.FNeg      -> per_component_builder L.build_fneg e'
␣␣␣␣␣␣␣␣␣␣|␣SA.BNot␣␣␣␣␣␣␣->␣per_component_builder␣L.build_not␣e'
         | SA.Int2Float ->
            per_component_builder (fun e -> L.build_sitofp e f32_t) e'
␣␣␣␣␣␣␣␣␣␣|␣SA.Float2Int␣␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣per_component_builder␣(fun␣e␣->␣L.build_fptosi␣e␣i32_t)␣e'
         | SA.Bool2Int  ->
            per_component_builder (fun e -> L.build_zext e i32_t) e'
␣␣␣␣␣␣␣␣␣␣|␣SA.Bool2Float␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣per_component_builder␣(fun␣e␣->␣L.build_uitofp␣e␣f32_t)␣e'
         | SA.Int2Bool  ->
            per_component_builder (L.build_icmp L.Icmp.Ne izero) e'
␣␣␣␣␣␣␣␣␣␣|␣SA.Float2Bool␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣per_component_builder
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(L.build_fcmp␣L.Fcmp.One␣(L.const_float␣f32_t␣0.))␣e')
         "tmp" builder
      | SA.STypeCons act ->
         match fst sexpr with
            A.Mat(_, _, _) | A.Array(_, Some _) ->
              fst (List.fold_left (fun (agg, idx) e ->
                let e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(L.build_insertvalue␣agg␣e' idx "tmp" builder, idx + 1))
            ((L.undef (ltype_of_typ (fst sexpr))), 0) act)
           | A.Array(t, None) -> let s = expr builder (List.hd act) in
            let a = L.undef (ltype_of_typ (fst sexpr)) in
            let a = L.build_insertvalue a s 0 "" builder in
            L.build_insertvalue a (L.build_array_malloc
              (ltype_of_typ t) s "" builder) 1 "" builder
           | A.Buffer(_) -> L.build_call create_buffer_func [| |] ""
              builder
           | A.Pipeline(p) ->
              let depth = List.hd act in
              let depth'␣=
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣L.build_zext␣(expr␣builder␣depth)␣i32_t␣""␣builder
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣pdecl␣=␣StringMap.find␣p␣pipeline_decls␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣fshader␣=␣StringMap.find␣pdecl.SA.sfshader␣
    shader_globals␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣vshader␣=␣StringMap.find␣pdecl.SA.svshader␣
    shader_globals␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣tmp␣=␣L.build_alloca␣pipeline_t␣"pipeline_tmp"␣builder␣
    in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣v␣=␣L.build_gep␣vshader␣[|␣izero;␣izero␣|]␣""␣builder␣
    in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣f␣=␣L.build_gep␣fshader␣[|␣izero;␣izero␣|]␣""␣builder␣
    in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ignore
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(L.build_call␣create_pipeline_func␣[|␣tmp;␣v;␣f;␣depth'
    |] "" builder);
              L.build_load tmp "" builder
           | A.Window ->
              (match act with
                  [w; h; offscreen] ->
                    let w'␣=␣expr␣builder␣w␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣h' = expr builder h in
                    let offscreen'␣=
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣L.build_zext␣(expr␣builder␣offscreen)␣i32_t␣""␣
    builder
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣L.build_call␣create_window_func
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣[|␣w'; h';␣offscreen' |] "" builder
                | _ -> raise (Failure "shouldn't␣get␣here"))
           | _ -> raise (Failure "shouldn't␣get␣here")

    in

    let copy_out_params builder =
      List.iter2 (fun p (q, (_, n)) ->
        if q <> A.In then
          let tmp = L.build_load (StringMap.find n formals) "" builder in
          ignore (L.build_store tmp p builder))
      (Array.to_list (L.params the_function)) fdecl.SA.sformals
    in

    (* Build a list of statments, and invoke "f␣builder" if the list doesn't
␣␣␣␣␣*␣end␣with␣a␣branch␣instruction␣(break,␣continue,␣return)␣*)
␣␣␣␣let␣rec␣stmts␣break_bb␣continue_bb␣builder␣sl␣f␣=
␣␣␣␣␣␣let␣builder␣=␣List.fold_left␣(stmt␣break_bb␣continue_bb)␣builder␣sl␣
    in
␣␣␣␣␣␣match␣L.block_terminator␣(L.insertion_block␣builder)␣with
␣␣␣␣Some␣_␣->␣()
␣␣␣␣␣␣|␣None␣->␣ignore␣(f␣builder)
␣␣␣␣(*␣Build␣the␣code␣for␣the␣given␣statement;␣return␣the␣builder␣for
```

```
␣␣␣␣␣␣␣the␣statement's successor *)
    and stmt break_bb continue_bb builder = function
        SA.SAssign (lval, e) -> handle_assign builder lval e; builder
      | SA.SCall (_, "print", [e]) ->
          let e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_call␣print_string_func␣[|␣e' |] "" builder);
          builder
      | SA.SCall (_, "printi", [e]) | SA.SCall (_, "printb", [e]) ->
      ignore
            (L.build_call printf_func [| int_format_str ; (expr builder e)
                |]
            "printf" builder);
          builder
      | SA.SCall (_, "printf", [e]) ->
      ignore
            (L.build_call printf_func
              [| float_format_str ;
                 L.build_fpext (expr builder e) f64_t "tmp" builder |]
              "printf" builder);
          builder
       | SA.SCall (ret, "sin", [e]) ->
          let e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣let␣ret␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣sin_e␣=
␣␣␣␣␣␣␣␣␣␣␣␣L.build_call␣sin_func␣[|␣e' |]
              "sin" builder
          in
      ignore (L.build_store sin_e ret builder); builder
       | SA.SCall (ret, "cos", [e]) ->
          let e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣let␣ret␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣cos_e␣=
␣␣␣␣␣␣␣␣␣␣␣␣L.build_call␣cos_func␣[|␣e' |]
              "cos" builder
          in
      ignore (L.build_store cos_e ret builder); builder
       | SA.SCall (ret, "sqrt", [e]) ->
          let e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣let␣ret␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣sqrt_e␣=
␣␣␣␣␣␣␣␣␣␣␣␣L.build_call␣sqrt_func␣[|␣e' |]
              "sqrt" builder
          in
      ignore (L.build_store sqrt_e ret builder); builder
        | SA.SCall (ret, "floor", [e]) ->
          let e'␣=␣expr␣builder␣e␣in
␣␣␣␣␣␣␣␣␣␣let␣ret␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣floor_e␣=
```

```
␣␣␣␣␣␣␣␣␣␣␣␣L.build_call␣floor_func␣[|␣e' |]
             "floor" builder
         in
      ignore (L.build_store floor_e ret builder); builder
       | SA.SCall (ret, "pow", [base; power]) ->
         let base'␣=␣expr␣builder␣base␣in
␣␣␣␣␣␣␣␣␣␣let␣power' = expr builder power in
         let ret = lvalue builder ret in
         let pow_base_power =
           L.build_call pow_func [| base';␣power' |]
             "pow" builder
         in
      ignore (L.build_store pow_base_power ret builder); builder
      | SA.SCall (_, "printc", [e]) ->
         ignore
           (L.build_call printf_func [| char_format_str; (expr builder e)
               |]
             "printf" builder);
         builder
      | SA.SCall (_, "set_active_window", [w]) ->
         ignore (L.build_call set_active_window_func [| expr builder w |]
             ""
           builder);
         builder
      | SA.SCall (_, "upload_buffer", [buf; data]) ->
         let buf'␣=␣expr␣builder␣buf␣in
␣␣␣␣␣␣␣␣␣␣let␣data', size = (match (fst data) with
           A.Array(A.Mat(_, 1, s), Some n) ->
             (lvalue builder data, L.const_int i32_t (4 * s * n))
         | A.Array(A.Mat(_, 1, n), None) -> let s = expr builder data in
             (L.build_extractvalue s 1 "" builder,
             L.build_mul (L.const_int i32_t (4 * n))
             (L.build_extractvalue s 0 "" builder) "" builder)
         | _ -> raise (Failure "not␣supported")) in
         let data'␣=␣L.build_bitcast␣data' voidp_t "" builder in
         ignore (L.build_call upload_buffer_func
           [| buf';␣data'; size;
              L.const_int i32_t 0x88E4 (* GL_STATIC_DRAW *) |] "" builder);
         builder
      | SA.SCall (_, "clear", [c]) ->
         let c'␣=␣lvalue␣builder␣c␣in
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_call␣clear_func␣[|␣c' |] "" builder);
         builder
      | SA.SCall (_, "draw", [p; i]) ->
         let p'␣=␣lvalue␣builder␣p␣in
␣␣␣␣␣␣␣␣␣␣let␣i' = expr builder i in
         ignore (L.build_call draw_arrays_func [| p';␣i' |] "" builder);
         builder
```

```
      | SA.SCall (_, "swap_buffers", [w]) ->
         let w'␣=␣expr␣builder␣w␣in
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_call␣swap_buffers_func␣[|␣w' |] "" builder);
         builder
      | SA.SCall (ret, "get_key", [w; key]) ->
         let w'␣=␣expr␣builder␣w␣in
␣␣␣␣␣␣␣␣␣␣let␣key' = expr builder key in
         let status = L.build_call get_key_func [| w';␣key' |] "" builder
             in
         let ret'␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣status␣=␣L.build_icmp␣L.Icmp.Ne␣izero␣status␣""␣builder␣in
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_store␣status␣ret' builder); builder
      | SA.SCall (ret, "get_mouse_button", [w; button]) ->
         let w'␣=␣expr␣builder␣w␣in
␣␣␣␣␣␣␣␣␣␣let␣button' = expr builder button in
         let status = L.build_call get_mouse_func [| w';␣button' |] ""
             builder in
         let ret'␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣status␣=␣L.build_icmp␣L.Icmp.Ne␣izero␣status␣""␣builder␣in
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_store␣status␣ret' builder); builder
      | SA.SCall (_, "get_mouse_pos", [w; x; y]) ->
         (* The GLFW function expects a pointer to a double for x and y,
             so we
          * have to convert double -> float ourselves after calling it.
          *)
         let w'␣=␣expr␣builder␣w␣in
␣␣␣␣␣␣␣␣␣␣let␣x' = lvalue builder x in
         let y'␣=␣lvalue␣builder␣y␣in
␣␣␣␣␣␣␣␣␣␣let␣tmp_x␣=␣L.build_alloca␣f64_t␣""␣builder␣in
␣␣␣␣␣␣␣␣␣␣let␣tmp_y␣=␣L.build_alloca␣f64_t␣""␣builder␣in
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_call␣get_mouse_pos_func␣[|␣w'; tmp_x; tmp_y |]
    "" builder);
         let out_x = L.build_fptrunc
           (L.build_load tmp_x "" builder) f32_t "" builder in
         let out_y = L.build_fptrunc
           (L.build_load tmp_y "" builder) f32_t "" builder in
         ignore (L.build_store out_x x'␣builder);
␣␣␣␣␣␣␣␣␣␣ignore␣(L.build_store␣out_y␣y' builder);
         builder
      | SA.SCall (_, "poll_events", []) ->
         ignore (L.build_call poll_events_func [| |] "" builder);
         builder
      | SA.SCall (ret, "window_should_close", [w]) ->
         let w'␣=␣expr␣builder␣w␣in
␣␣␣␣␣␣␣␣␣␣let␣ret␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣llret␣=␣L.build_icmp␣L.Icmp.Ne
␣␣␣␣␣␣␣␣␣␣␣␣(L.build_call␣should_close_func␣[|␣w' |] "" builder)
           izero "" builder in
```

```
        ignore (L.build_store llret ret builder); builder
    | SA.SCall (ret, "read_pixel", [x; y]) ->
        let x'␣=␣expr␣builder␣x␣in
␣␣␣␣␣␣␣␣␣␣let␣y' = expr builder y in
        let ret = lvalue builder ret in
        ignore (L.build_call read_pixel_func [| x';␣y'; ret |] ""
            builder);
        builder
    | SA.SCall (ret, "length", [arr]) ->
        let arr'␣=␣expr␣builder␣arr␣in
␣␣␣␣␣␣␣␣␣␣let␣ret␣=␣lvalue␣builder␣ret␣in
␣␣␣␣␣␣␣␣␣␣let␣len␣=␣(match␣fst␣arr␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣A.Array(_,␣Some␣len)␣->␣L.const_int␣i32_t␣len
␣␣␣␣␣␣␣␣␣␣␣␣|␣A.Array(_,␣None)␣->␣L.build_extractvalue␣arr' 0 "" builder
          | _ -> raise (Failure "unexpected␣type")) in
        ignore (L.build_store len ret builder); builder
    | SA.SCall (ret, "read_file", [path]) ->
        let path = expr builder path in
        let ret = lvalue builder ret in
        ignore (L.build_call read_file_func [| ret; path |] "" builder);
        builder
    | SA.SCall (ret, f, act) ->
       let (fdef, fdecl) = StringMap.find f function_decls in
   let actuals = (List.map2 (fun (q, (_, _)) e ->
        if q = A.In then expr builder e
        else lvalue builder e) fdecl.SA.sformals act) in
   let result = (match fdecl.SA.styp with A.Void -> ""
                                  | _ -> f ^ "_result") in
       let llret = L.build_call fdef (Array.of_list actuals) result
           builder in
       (match ret with
          (A.Void, SA.SNoexpr) -> ()
        | _ -> let ret = lvalue builder ret in
            ignore (L.build_store llret ret builder)
        ); builder
    | SA.SReturn e -> copy_out_params builder;
        ignore (match fdecl.SA.styp with
    A.Void -> L.build_ret_void builder
  | _ -> L.build_ret (expr builder e) builder); builder
    | SA.SBreak -> ignore (L.build_br break_bb builder); builder
    | SA.SContinue -> ignore (L.build_br continue_bb builder); builder
    | SA.SIf (predicate, then_stmts, else_stmts) ->
       let bool_val = expr builder predicate in
   let merge_bb = L.append_block context "merge" the_function in

   let then_bb = L.append_block context "then" the_function in
   stmts break_bb continue_bb (L.builder_at_end context then_bb)
       then_stmts
```

```
     (L.build_br merge_bb);

    let else_bb = L.append_block context "else" the_function in
    stmts break_bb continue_bb (L.builder_at_end context else_bb)
        else_stmts
     (L.build_br merge_bb);

    ignore (L.build_cond_br bool_val then_bb else_bb builder);
    L.builder_at_end context merge_bb

     | SA.SLoop (body, continue) ->
         let body_bb = L.append_block context "loop_body" the_function in
         let continue_bb = L.append_block context "loop_continue"
             the_function in
         let merge_bb = L.append_block context "loop_merge" the_function in

         ignore (L.build_br body_bb builder);

         let body_builder = L.builder_at_end context body_bb in
         stmts merge_bb continue_bb body_builder body
           (L.build_br continue_bb);

         let continue_builder = L.builder_at_end context continue_bb in
         stmts merge_bb continue_bb continue_builder continue
           (L.build_br body_bb);

         L.builder_at_end context merge_bb
   in

   if fdecl.SA.sfname = "main" then
     ignore (L.build_call init_func [| |] "" builder)
   else
     ()
   ;

   (* Build the code for each statement in the function *)
   let dummy_bb = L.append_block context "dummy" the_function in
   ignore (L.build_unreachable (L.builder_at_end context dummy_bb));
   stmts dummy_bb dummy_bb builder fdecl.SA.sbody
     (* Add a return if the last block falls off the end. Semantic checking
      * ensures that only functions that return void hit this path. *)
     (fun builder -> copy_out_params builder; L.build_ret_void builder)

 in

 List.iter build_function_body functions;
 the_module
```

prelude.blis

```
/* This file is included before any program compiled by the Blis compiler.
 * It is used to implement all the built-in functions that can be
    implemented
 * in Blis, instead of using a C library or GLSL built-in function.
 */

/* Keyboard & mouse constants copied from the GLFW header */

/* Printable keys */
const int KEY_SPACE =           32;
const int KEY_APOSTROPHE =      39;  /* ' */
const int KEY_COMMA =           44;  /* , */
const int KEY_MINUS =           45;  /* - */
const int KEY_PERIOD =          46;  /* . */
const int KEY_SLASH =           47;  /* / */
const int KEY_0 =               48;
const int KEY_1 =               49;
const int KEY_2 =               50;
const int KEY_3 =               51;
const int KEY_4 =               52;
const int KEY_5 =               53;
const int KEY_6 =               54;
const int KEY_7 =               55;
const int KEY_8 =               56;
const int KEY_9 =               57;
const int KEY_SEMICOLON =       59;  /* ; */
const int KEY_EQUAL =           61;  /* = */
const int KEY_A =               65;
const int KEY_B =               66;
const int KEY_C =               67;
const int KEY_D =               68;
const int KEY_E =               69;
const int KEY_F =               70;
const int KEY_G =               71;
const int KEY_H =               72;
const int KEY_I =               73;
const int KEY_J =               74;
const int KEY_K =               75;
const int KEY_L =               76;
const int KEY_M =               77;
const int KEY_N =               78;
const int KEY_O =               79;
const int KEY_P =               80;
const int KEY_Q =               81;
const int KEY_R =               82;
const int KEY_S =               83;
```

```
const int KEY_T =               84;
const int KEY_U =               85;
const int KEY_V =               86;
const int KEY_W =               87;
const int KEY_X =               88;
const int KEY_Y =               89;
const int KEY_Z =               90;
const int KEY_LEFT_BRACKET =    91;  /* [ */
const int KEY_BACKSLASH =       92;  /* \ */
const int KEY_RIGHT_BRACKET =   93;  /* ] */
const int KEY_GRAVE_ACCENT =    96;  /* ` */
const int KEY_WORLD_1 =         161; /* non-US #1 */
const int KEY_WORLD_2 =         162; /* non-US #2 */

/* Function keys */
const int KEY_ESCAPE =          256;
const int KEY_ENTER =           257;
const int KEY_TAB =             258;
const int KEY_BACKSPACE =       259;
const int KEY_INSERT =          260;
const int KEY_DELETE =          261;
const int KEY_RIGHT =           262;
const int KEY_LEFT =            263;
const int KEY_DOWN =            264;
const int KEY_UP =              265;
const int KEY_PAGE_UP =         266;
const int KEY_PAGE_DOWN =       267;
const int KEY_HOME =            268;
const int KEY_END =             269;
const int KEY_CAPS_LOCK =       280;
const int KEY_SCROLL_LOCK =     281;
const int KEY_NUM_LOCK =        282;
const int KEY_PRINT_SCREEN =    283;
const int KEY_PAUSE =           284;
const int KEY_F1 =              290;
const int KEY_F2 =              291;
const int KEY_F3 =              292;
const int KEY_F4 =              293;
const int KEY_F5 =              294;
const int KEY_F6 =              295;
const int KEY_F7 =              296;
const int KEY_F8 =              297;
const int KEY_F9 =              298;
const int KEY_F10 =             299;
const int KEY_F11 =             300;
const int KEY_F12 =             301;
const int KEY_F13 =             302;
const int KEY_F14 =             303;
```

```
const int KEY_F15 =            304;
const int KEY_F16 =            305;
const int KEY_F17 =            306;
const int KEY_F18 =            307;
const int KEY_F19 =            308;
const int KEY_F20 =            309;
const int KEY_F21 =            310;
const int KEY_F22 =            311;
const int KEY_F23 =            312;
const int KEY_F24 =            313;
const int KEY_F25 =            314;
const int KEY_KP_0 =           320;
const int KEY_KP_1 =           321;
const int KEY_KP_2 =           322;
const int KEY_KP_3 =           323;
const int KEY_KP_4 =           324;
const int KEY_KP_5 =           325;
const int KEY_KP_6 =           326;
const int KEY_KP_7 =           327;
const int KEY_KP_8 =           328;
const int KEY_KP_9 =           329;
const int KEY_KP_DECIMAL =     330;
const int KEY_KP_DIVIDE =      331;
const int KEY_KP_MULTIPLY =    332;
const int KEY_KP_SUBTRACT =    333;
const int KEY_KP_ADD =         334;
const int KEY_KP_ENTER =       335;
const int KEY_KP_EQUAL =       336;
const int KEY_LEFT_SHIFT =     340;
const int KEY_LEFT_CONTROL =   341;
const int KEY_LEFT_ALT =       342;
const int KEY_LEFT_SUPER =     343;
const int KEY_RIGHT_SHIFT =    344;
const int KEY_RIGHT_CONTROL =  345;
const int KEY_RIGHT_ALT =      346;
const int KEY_RIGHT_SUPER =    347;
const int KEY_MENU =           348;

const int KEY_LAST =           348;

const int MOUSE_BUTTON_LEFT =  0;
const int MOUSE_BUTTON_RIGHT = 1;
const int MOUSE_BUTTON_MIDDLE = 2;

int char_to_digit(u8 digit) {
  if (digit == '0') {
    return 0;
  } else if (digit == '1') {
```

```
    return 1;
  } else if (digit == '2') {
    return 2;
  } else if (digit == '3') {
    return 3;
  } else if (digit == '4') {
    return 4;
  } else if (digit == '5') {
    return 5;
  } else if (digit == '6') {
    return 6;
  } else if (digit == '7') {
    return 7;
  } else if (digit == '8') {
    return 8;
  } else if (digit == '9') {
    return 9;
  }

  return -1;
}

float char_to_digitf(u8 digit)
{
  return float(char_to_digit(digit));
}

// Converts a substring into an integer.
//
// Inputs:
//   string: u8[] containing the substring to be parsed
//   start: the index of the first character of the substring
//   end: the index of the last character of the substring
// Outputs:
//   result: the integer represented by the substring
//   returns: true if successful, false otherwise
bool substring_to_integer(u8[] string, int start, int end, out int result)
    {
  //reject nonsensical input
  int str_length = length(string);
  if (str_length == 0 || end >= str_length || start < 0 || start > end) {
    return false;
  }
  result = 0;
  int i;
  for (i = start; i <= end; i = i + 1) {
    int digit = char_to_digit(string[i]);
    if (digit < 0) {
```

```
      return false;
    }
    //compute the place value of this digit
    int place_value = 1;
    int j;
    for (j = 0; j < end - i; j = j + 1) {
      place_value = place_value * 10;
    }
    result = result + digit * place_value;
  }
  return true;
}

bool string_to_integer(u8[] string, out int result)
{
  return substring_to_integer(string, 0, length(string) - 1, result);
}

// Converts a substring into a float.
// The substring must contain a decimal point and can have a leading '-'
    symbol.
// It may not contain e or E.
//
// Inputs:
//   string: u8[] containing the substring to be parsed
//   start: the index of the first character of the substring
//   end: the index of the last character of the substring
// Output:
//   result: the float represented by the substring
//   returns: true if successful, false otherwise
bool substring_to_float(u8[] string, int start, int end, out float result)
    {
  //reject nonsensical input
  int str_length = length(string);
  if (str_length == 0 || end >= str_length || start < 0 || start >= end) {
    return false;
  }
  //check for a leading '-' symbol
  float sign = 1.0;
  if (string[start] == '-') {
    sign = -1.0;
    start = start + 1;
    if (start == end) {
      return false;
    }
  }
  //determine the position of the decimal point
  int point = -1;
```

```
int i;
for (i = start; i <= end; i = i + 1) {
  if (string[i] == '.') {
    if (point == -1) {
      point = i;
    } else {
      //error: contains multiple decimal points
      return false;
    }
  }
}
if (point == -1) {
  //error: no decimal point found
  return false;
}
result = 0.;
//compute the integer part
if (point > start) {
  int i;
  for (i = start; i < point; i = i + 1) {
    float digit = char_to_digitf(string[i]);
    if (digit < 0.) {
      return false;
    }
    float place_value = 1.;
    int j;
    for (j = 0; j < point - 1 - i; j = j + 1) {
      place_value = place_value * 10.;
    }
    result = result + digit * place_value;
  }
}
//compute the decimal part
if (point < end) {
  int i;
  for (i = point + 1; i <= end; i = i + 1) {
    float digit = char_to_digitf(string[i]);
    if (digit < 0.) {
      return false;
    }
    float place_value = 0.1;
    int j;
    for ( j = 0; j < i - (point + 1); j = j + 1) {
      place_value = place_value / 10.0;
    }
    result = result + digit * place_value;
  }
}
```

```
  result = sign * result;
  return true;
}

bool string_to_float(u8[] string, out float result)
{
  return substring_to_float(string, 0, length(string) - 1, result);
}

// split a string into substrings separated by "sep". For example,
// "aababba" with sep = 'a' returns ["b", "bb"].
u8[][] split(u8[] string, u8 sep)
{
  int num_strings = 0;
  int cur_string_pos = 0;
  int i;

  // count how many strings there will be
  for (i = 0; i <= length(string); i = i + 1) {
    if (i == length(string) || string[i] == sep) {
      if (cur_string_pos != 0)
        num_strings = num_strings + 1;
      cur_string_pos = 0;
    } else {
      cur_string_pos = cur_string_pos + 1;
    }
  }

  u8[][] strings = u8[][](num_strings);
  int cur_string = 0;
  u8[] string_buffer = u8[](4);

  for (i = 0; i <= length(string); i = i + 1) {
    if (i == length(string) || string[i] == sep) {
      if (cur_string_pos != 0) {
        strings[cur_string] = u8[](cur_string_pos);
        int j;
        for (j = 0; j < cur_string_pos; j = j + 1)
          strings[cur_string][j] = string_buffer[j];
        cur_string = cur_string + 1;
      }
      cur_string_pos = 0;
    } else {
      string_buffer[cur_string_pos] = string[i];
      cur_string_pos = cur_string_pos + 1;
      if (cur_string_pos >= length(string_buffer)) {
        // expand string_buffer
        u8[] new_string_buffer = u8[](length(string_buffer) * 2);
```

```
      int j;
      for (j = 0; j < cur_string_pos; j = j + 1)
        new_string_buffer[j] = string_buffer[j];
      string_buffer = new_string_buffer;
    }
  }
}

  return strings;
}

// Parses an obj file such that:
//   Lines starting with # and blank lines are ignored.
//   Lines starting with v describe vertices. E.g. v -1.1 43. -.123
//   Lines starting with f describe faces. E.g. f 0 1 2
// Inputs:
//   fpath: path to the obj file
//   numVertices: the number of vertices in the obj file
//   numFaces: the number of faces in the obj file
// Outputs:
//   verts: vec3[] of vertices
//   tris: int[] of face indices
//   returns: true if successful, false otherwise
bool read_obj(u8[] fpath, out vec3[] verts, out int[] tris)
{
  //read file
  u8[] file = read_file(fpath);

  u8[][] lines = split(file, '\n');

  int vcount = 0;
  int fcount = 0;
  int i;
  for (i = 0; i < length(lines); i = i + 1) {
    if (lines[i][0] == 'v')
      vcount = vcount + 1;
    else if (lines[i][0] == 'f')
      fcount = fcount + 1;
    else if (lines[i][0] != '#')
      return false;
  }

  verts = vec3[](vcount);
  tris = int[](3 * fcount);
  vcount = 0;
  fcount = 0;

  for (i = 0; i < length(lines); i = i + 1) {
```

```
    u8[][] tokens = split(lines[i], '␣');
    if (lines[i][0] == 'v') {
      if (length(tokens) != 4)
        return false;
      if (!string_to_float(tokens[1], verts[vcount].x))
        return false;
      if (!string_to_float(tokens[2], verts[vcount].y))
        return false;
      if (!string_to_float(tokens[3], verts[vcount].z))
        return false;
      vcount = vcount + 1;
    } else if (lines[i][0] == 'f') {
      if (length(tokens) != 4)
        return false;
      int tmp;
      if (!string_to_integer(tokens[1], tmp))
        return false;
      tris[3*fcount + 0] = tmp - 1;
      if (!string_to_integer(tokens[2], tmp))
        return false;
      tris[3*fcount + 1] = tmp - 1;
      if (!string_to_integer(tokens[3], tmp))
        return false;
      tris[3*fcount + 2] = tmp - 1;
      fcount = fcount + 1;
    }
  }

  return true;
}

@gpu float deg_to_rad(float x)
{
  return x * 0.017453293;
}

@gpu float tan(float x)
{
  return sin(x) / cos(x);
}

@gpu float dot3(vec3 a, vec3 b)
{
  return a.x * b.x + a.y * b.y + a.z * b.z;
}

@gpu vec3 cross(vec3 a, vec3 b)
{
```

```
  return vec3(a.y * b.z - a.z * b.y,
              a.z * b.x - a.x * b.z,
              a.x * b.y - a.y * b.x);
}

@gpu float norm3(vec3 x)
{
  return sqrt(dot3(x, x));
}

@gpu vec3 normalize3(vec3 x)
{
  return x * (1.0 / norm3(x));
}

// functions to build transformation matrices

@gpu mat4x4 identity()
{
  return mat4x4(vec4(1., 0., 0., 0.),
                vec4(0., 1., 0., 0.),
                vec4(0., 0., 1., 0.),
                vec4(0., 0., 0., 1.));
}

@gpu mat4x4 scale_x(float scale)
{
  mat4x4 ret = identity();
  ret.x.x = scale;
  return ret;
}

@gpu mat4x4 scale_y(float scale)
{
  mat4x4 ret = identity();
  ret.y.y = scale;
  return ret;
}

@gpu mat4x4 scale_z(float scale)
{
  mat4x4 ret = identity();
  ret.z.z = scale;
  return ret;
}

@gpu mat4x4 rotate_x(float theta)
{
```

```
  mat4x4 ret = identity();

  ret.z.z = ret.y.y = cos(theta);
  ret.y.z = sin(theta);
  ret.z.y = -ret.y.z;
  return ret;
}

@gpu mat4x4 rotate_y(float theta)
{
  mat4x4 ret = identity();

  ret.x.x = ret.z.z = cos(theta);
  ret.z.x = sin(theta);
  ret.x.z = -ret.z.x;
  return ret;
}

@gpu mat4x4 rotate_z(float theta)
{
  mat4x4 ret = identity();

  ret.x.x = ret.y.y = cos(theta);
  ret.x.y = sin(theta);
  ret.y.x = -ret.x.y;
  return ret;
}

@gpu mat4x4 translate(vec3 offset)
{
  mat4x4 ret = identity();
  ret.w.x = offset.x;
  ret.w.y = offset.y;
  ret.w.z = offset.z;
  return ret;
}

mat4x4 perspective(float fovy, float aspect,
                   float near, float far)
{
  float top  = tan(deg_to_rad(fovy) / 2.) * near;
  float right = top * aspect;

  mat4x4 ret = identity();
  ret.x.x = near / right;
  ret.y.y = near / top;
  ret.z.z = -(far + near) / (far - near);
  ret.w.z = -2.0 * far * near / (far - near);
```

```
  ret.z.w = -1.0;
  return ret;
}

mat4x4 look_at(vec3 eye, vec3 at, vec3 up)
{
  vec3 n = normalize3(eye - at);
  vec3 u = normalize3(cross(up, n));
  vec3 v = normalize3(cross(n, u));
  return mat4x4(vec4(u.x, v.x, n.x, 0.0),
                vec4(u.y, v.y, n.y, 0.0),
                vec4(u.z, v.z, n.z, 0.0),
                vec4(0.0, 0.0, 0.0, 1.0)) * translate(-eye);
}
```

glslcodegen.ml

```
(* Code generation for shaders. Here we take each entrypoint and turn it
    into a
 * GLSL shader.
 *)

module A = Ast
module SA = Sast

module StringSet = Set.Make(String)
module StringMap = Map.Make(String)

type symbol_table = {
  scope : string StringMap.t;
  used_names : StringSet.t;
}

let empty_table = {
  scope = StringMap.empty;
  used_names = StringSet.empty;
}

type translation_environment = {
  table : symbol_table;
  cur_qualifier : A.func_qualifier;
  forloop_update_statement : string;
}

(* return a fresh name given a set of already-used names. The original
    name is
 * usually a prefix of the new name, although this may not be true if the
 * original name is reserved by OpenGL.
```

```
 *
 *)
let add_symbol_table table orig =
  (* copied from the GLSL 3.30 spec. The weird line wrapping is identical
      to the
   * PDF to ease comparisons.
   *)
  let glsl_keywords = [
    "attribute"; "const"; "uniform"; "varying";
    "layout";
    "centroid"; "flat"; "smooth"; "noperspective";
    "break"; "continue"; "do"; "for"; "while"; "switch"; "case"; "default";
    "if"; "else";
    "in"; "out"; "inout";
    "float"; "int"; "void"; "bool"; "true"; "false";
    "invariant";
    "discard"; "return";
    "mat2"; "mat3"; "mat4";
    "mat2x2"; "mat2x3"; "mat2x4";
    "mat3x2"; "mat3x3"; "mat3x4";
    "mat4x2"; "mat4x3"; "mat4x4";
    "vec2"; "vec3"; "vec4"; "ivec2"; "ivec3"; "ivec4"; "bvec2"; "bvec3";
        "bvec4";
    "uint"; "uvec2"; "uvec3"; "uvec4";
    "lowp"; "mediump"; "highp"; "precision";
    "sampler1D"; "sampler2D"; "sampler3D"; "samplerCube";
    "sampler1DShadow"; "sampler2DShadow"; "samplerCubeShadow";
    "sampler1DArray"; "sampler2DArray";
    "sampler1DArrayShadow"; "sampler2DArrayShadow";
    "isampler1D"; "isampler2D"; "isampler3D"; "isamplerCube";
    "isampler1DArray"; "isampler2DArray";
    "usampler1D"; "usampler2D"; "usampler3D"; "usamplerCube";
    "usampler1DArray"; "usampler2DArray";
    "sampler2DRect"; "sampler2DRectShadow"; "isampler2DRect";
        "usampler2DRect";
    "samplerBuffer"; "isamplerBuffer"; "usamplerBuffer";
    "sampler2DMS"; "isampler2DMS"; "usampler2DMS";
    "sampler2DMSArray"; "isampler2DMSArray"; "usampler2DMSArray";
    "struct";
    "common"; "partition"; "archive";
    "asm";
    "class"; "union"; "enum"; "typedef"; "template"; "this"; "packed";
    "goto";
    "inline"; "noinline"; "volatile"; "public"; "static"; "extern";
        "external"; "interface";
    "long"; "short"; "double"; "half"; "fixed"; "unsigned"; "superp";
    "input"; "output";
    "hvec2"; "hvec3"; "hvec4"; "dvec2"; "dvec3"; "dvec4"; "fvec2"; "fvec3";
```

```
        "fvec4";
    "sampler3DRect";
    "filter";
    "image1D"; "image2D"; "image3D"; "imageCube";
    "iimage1D"; "iimage2D"; "iimage3D"; "iimageCube";
    "uimage1D"; "uimage2D"; "uimage3D"; "uimageCube";
    "image1DArray"; "image2DArray";
    "iimage1DArray"; "iimage2DArray"; "uimage1DArray"; "uimage2DArray";
    "image1DShadow"; "image2DShadow";
    "image1DArrayShadow"; "image2DArrayShadow";
    "imageBuffer"; "iimageBuffer"; "uimageBuffer";
    "sizeof"; "cast";
    "namespace"; "using";
    "row_major";
    (* built-in functions in GLSL *)
    "sin"; "cos"; "pow"; "sqrt"; "floor";
    "radians"; "degrees"; "tan"; "asin"; "acos"; "atan";
    "sinh"; "cosh"; "tanh"; "asinh"; "acosh"; "atanh";
    "exp"; "log"; "exp2"; "log2"; "inversesqrt"; "abs";
     "sign"; "trunc"; "round"; "roundEven"; "ceil";
     "fract"; "mod"; "modf"; "min"; "max"; "clamp";
     "mix"; "step"; "smoothstep"; "isnan"; "isinf";
     "floatBitsToInt"; "floatBitsToUint"; "intBitsToFloat";
     "uintBitsToFloat"; "length"; "distance"; "dot"; "cross";
     "normalize"; "ftransform"; "faceforward"; "reflect";
     "refract"; "matrixCompMult"; "outerProduct";
     "transpose"; "determinant"; "inverse";
     "lessThan"; "lessThanEqual"; "greaterThan";
     "greaterThanEqual"; "equal"; "notEqual";
     "any"; "all"; "not"; "textureSize"; "texture";
     "textureProj"; "textureLod"; "textureOffset";
     "texelFetch"; "texelFetchOffset"; "textureProjOffset";
     "textureLodOffset"; "textureProjLod"; "textureProjLodOffset";
     "textureGrad"; "textureGradOffset"; "textureProjGrad";
     "textureProjGradOffset"; "texture1D"; "texture1DProj";
     "texture1DLod"; "texture1DProjLod"; "texture2D"; "texture2DProj";
     "texture2DLod"; "texture2DProjLod"; "texture3D"; "texture3DProj";
     "texture3DLod"; "texture3DProjLod"; "textureCube";
     "textureCubeLod"; "shadow1D"; "shadow2D"; "shadow1DProj";
     "shadow2DProj"; "shadow1DLod"; "shadow2DLod";
     "shadow1DProjLod"; "shadow2DProjLod"; "dFdx";
     "dFdy"; "fwidth"; "noise1"; "noise2"; "noise3";
     "noise4"; "EmitVertex"; "EndPrimitive"; ]
  in

  (* names starting with "gl_" are reserved in GLSL *)
  let orig'␣=␣if␣String.length␣orig␣>␣3␣&&␣String.sub␣orig␣0␣3␣=␣"gl_"␣then
␣␣␣␣String.sub␣orig␣3␣(String.length␣orig␣-␣3)
```

```
␣␣else
␣␣␣␣orig
␣␣in
␣␣(*␣avoid␣using␣GLSL␣keywords␣*)
␣␣let␣orig’ = if List.mem orig’␣glsl_keywords␣then
␣␣␣␣orig’ ^ "_"
  else
    orig
  in
  (* if we wind up with an empty name, either because the caller passed in
      one
   * or there was a name called "gl_" in Blis, then we can’t␣return␣it,␣so␣
       just
␣␣␣*␣change␣it␣to␣"_"
␣␣␣*)
␣␣let␣orig’ = if orig’␣=␣""␣then␣"_"␣else␣orig’ in
  let orig’␣=␣if␣not␣(StringSet.mem␣orig’ table.used_names) then
    orig’
␣␣else
␣␣␣␣(*␣keep␣appending␣digits␣until␣we␣get␣a␣new␣name␣*)
␣␣␣␣let␣rec␣get_name␣orig␣n␣=
␣␣␣␣␣␣let␣orig’ = orig ^ string_of_int n in
        if not (StringSet.mem orig’␣table.used_names)␣then
␣␣␣␣␣␣␣␣␣␣orig’
        else
          get_name orig (n + 1)
    in
    get_name orig’␣0
␣␣in
␣␣({␣scope␣=␣StringMap.add␣orig␣orig’ table.scope;
    used_names = StringSet.add orig’␣table.used_names␣},␣orig’)

let add_variable_name env name =
  let table’,␣new_name␣=␣add_symbol_table␣env.table␣name␣in
␣␣({␣env␣with␣table␣=␣table’ }, new_name)

let translate ((structs, _, globals, functions) : SA.sprogram) =
  let env = {
    table = empty_table;
    cur_qualifier = A.Both;
    forloop_update_statement = "";
  }
  in

  (* create mapping from const global to initializer *)
  let global_map = List.fold_left (fun map (_, (_, n), e) ->
    match e with
        Some e -> StringMap.add n e map
```

```
    | None -> map) StringMap.empty globals
in

(* structs and functions share a namespace in GLSL, so use the same
    table to
 * translate the names for them.
 *)
let struct_table =
  List.fold_left (fun table sdecl ->
    fst (add_symbol_table table sdecl.A.sname))
  { used_names = StringSet.singleton "dummy_struct";
    scope = StringMap.empty }
  structs
in

let func_table =
  List.fold_left (fun table fdecl ->
    fst (add_symbol_table table fdecl.SA.sfname))
  struct_table functions
in

(* returns the GLSL type for the Blis type stripped of array-ness
 * for example, returns "vec4" for vec4[10][2]
 *)
let rec string_of_base_typ = function
    A.Mat(A.Int, 1, 1) | A.Mat(A.Byte, 1, 1) -> "int"
  | A.Mat(A.Float, 1, 1) -> "float"
  | A.Mat(A.Bool, 1, 1) -> "bool"
  | A.Mat(A.Int, 1, l) | A.Mat(A.Byte, 1, l) -> "ivec" ^ string_of_int l
  | A.Mat(A.Float, 1, l) -> "vec" ^ string_of_int l
  | A.Mat(A.Bool, 1, l) -> "bvec" ^ string_of_int l
  | A.Mat(A.Float, w, 1) -> "vec" ^ string_of_int w
  | A.Mat(A.Float, w, l) -> "mat" ^ string_of_int w ^ "x" ^ string_of_int
      l
  | A.Mat(_, _, _) -> raise (Failure "unimplemented")
  | A.Struct(name) -> StringMap.find name struct_table.scope
  | A.Buffer(_) -> "dummy_struct"
  | A.Window -> "dummy_struct"
  | A.Pipeline(_) -> "dummy_struct"
  | A.Void -> "void"
  | A.Array(typ, _) -> string_of_base_typ typ
in


(* returns the arrays required for a given Blis type
 * for example, returns "[10][2]" for vec4[10][2]
 *)
let rec string_of_array = function
```

```
    A.Array(typ, n) -> "[" ^
      (match n with Some(w) -> string_of_int w | _ -> "0") ^ "]"
      ^ string_of_array typ
  | _ -> ""
in

let string_of_typ typ = string_of_base_typ typ ^ string_of_array typ
in

let string_of_bind env (typ, name) =
  let env, new_name = add_variable_name env name in
  (env, string_of_base_typ typ ^ "␣" ^ new_name ^ string_of_array typ)
in

let struct_members = List.fold_left (fun map sdecl ->
  let table = List.fold_left (fun table (_, name) ->
    fst (add_symbol_table table name)) empty_table sdecl.A.members
  in
  StringMap.add sdecl.A.sname table map)
StringMap.empty structs
in

let glsl_structs = String.concat "\n\n"
  (List.map (fun sdecl ->
    let members = StringMap.find sdecl.A.sname struct_members in
    let glsl_name = StringMap.find sdecl.A.sname struct_table.scope in
    "struct␣" ^ glsl_name ^ "␣{\n" ^
    (String.concat "\n" (List.map (fun (typ, name) ->
      let glsl_name = StringMap.find name members.scope in
      string_of_base_typ typ ^ "␣" ^ glsl_name ^ string_of_array typ ^
          ";")
    sdecl.A.members)) ^
    "\n};") structs) ^ "\n\n"
in

(* from
    http://stackoverflow.com/questions/10068713/string-to-list-of-char *)
let explode s =
  let rec exp i l =
    if i < 0 then l else exp (i - 1) (s.[i] :: l) in
  exp (String.length s - 1) []
in

let rec translate_stmts env slist =
  let rec expr env (typ, e) = match e with
      SA.SIntLit(i) -> string_of_int i
    | SA.SFloatLit(f) -> string_of_float f
    | SA.SBoolLit(b) -> if b then "true" else "false"
```

```
      | SA.SCharLit(c) -> string_of_int (Char.code c)
      | SA.SStringLit(s) ->
          "int[" ^ string_of_int (String.length s) ^ "](" ^
            String.concat ",␣"
              (List.map (fun c -> string_of_int (Char.code c)) (explode s)) ^
            ")"
      | SA.SId(n) ->
          (try StringMap.find n env.table.scope
          with Not_found ->
            let e = StringMap.find n global_map in
            expr env e)
      | SA.SStructDeref(e, mem) -> let e'␣=␣expr␣env␣e␣in
␣␣␣␣␣␣␣␣␣␣(match␣fst␣e␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣A.Mat(_,␣1,␣_)␣->␣"("␣^␣e' ^ ")." ^ mem
            | A.Mat(_, _, _) -> "(" ^ e'␣^␣")["␣^␣(match␣mem␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"x"␣->␣"0"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣"y"␣->␣"1"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣"z"␣->␣"2"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣"w"␣->␣"3"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣_␣->␣raise␣(Failure␣"shouldln't get here"))␣^␣"]"
␣␣␣␣␣␣␣␣␣␣␣␣|␣A.Struct(name)␣->
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣members␣=␣StringMap.find␣name␣struct_members␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let␣glsl_mem␣=␣StringMap.find␣mem␣members.scope␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"("␣^␣e'␣^␣")."␣^␣glsl_mem
␣␣␣␣␣␣␣␣␣␣␣␣|␣_␣->␣raise␣(Failure␣"unimplemented"))
␣␣␣␣␣␣|␣SA.SArrayDeref(e,␣idx)␣->
␣␣␣␣␣␣␣␣␣␣"("␣^␣expr␣env␣e␣^␣")["␣^␣expr␣env␣idx␣^␣"]"
␣␣␣␣␣␣|␣SA.SBinop(e1,␣op,␣e2)␣->
␣␣␣␣␣␣␣␣␣␣let␣e1'␣=␣expr␣env␣e1␣in
␣␣␣␣␣␣␣␣␣␣let␣e2'␣=␣expr␣env␣e2␣in
␣␣␣␣␣␣␣␣␣␣let␣e1cols,␣e1rows,␣e2cols,␣e2rows␣=␣match␣fst␣e1,␣fst␣e2␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣A.Mat(_,␣w,␣l),␣A.Mat(_,␣w',␣l')␣->␣w,␣l,␣w',␣l'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣_␣->␣raise␣(Failure␣"shoudln't␣get␣here")
␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣␣␣␣␣let␣ordinary_binop␣str␣=
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"("␣^␣e1' ^ ")␣" ^ str ^ "␣(" ^ e2'␣^␣")"
␣␣␣␣␣␣␣␣␣␣in
␣␣␣␣␣␣␣␣␣␣let␣fmat_mult␣=
␣␣␣␣␣␣␣␣␣␣␣␣if␣(e1rows␣>␣1␣&&␣e2rows␣>␣1)␣||␣(e1rows␣=␣1␣&&␣e1cols␣=␣1)␣||
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(e2rows␣=␣1␣&&␣e2cols␣=␣1)␣then␣ordinary_binop␣"*"
␣␣␣␣␣␣␣␣␣␣␣␣else␣if␣e1rows␣=␣1␣&&␣e2cols␣=␣1␣then
␣␣␣␣␣␣␣␣␣␣␣␣␣␣"dot("␣^␣"("␣^␣e1' ^ ")␣," ^ "␣(" ^ e2'␣^␣")"␣^␣")␣"
␣␣␣␣␣␣␣␣␣␣␣␣else␣if␣e1cols␣=␣1␣&&␣e2rows␣=␣1␣then
␣␣␣␣␣␣␣␣␣␣␣␣␣␣"outerProduct("␣^␣"("␣^␣e1' ^ ")␣," ^ "␣(" ^ e2'␣^␣")"␣^␣")␣"
␣␣␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣Row␣vector␣times␣matrix␣*)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣"transpose(␣"␣^␣"␣("␣^␣e2' ^ ")" ^")␣" ^ "*" ^ "(" ^ e1'␣^␣
    ")"
```

```
          in
          (match op with
              SA.IAdd | SA.FAdd -> ordinary_binop "+"
            | SA.ISub | SA.FSub -> ordinary_binop "-"
            | SA.IMult | SA.FMult | SA.Splat -> ordinary_binop "*"
            | SA.IMod -> ordinary_binop "%"
            | SA.IDiv | SA.FDiv -> ordinary_binop "/"
            | SA.IEqual | SA.FEqual | SA.BEqual | SA.U8Equal -> 
    ordinary_binop "=="
            | SA.INeq | SA.FNeq | SA.BNeq | SA.U8Neq -> ordinary_binop "!="
            | SA.ILess | SA.FLess -> ordinary_binop "<"
            | SA.IGreater | SA.FGreater -> ordinary_binop ">"
            | SA.IGeq | SA.FGeq -> ordinary_binop ">="
            | SA.ILeq | SA.FLeq -> ordinary_binop "<="
            | SA.BAnd -> ordinary_binop "&&"
            | SA.BOr -> ordinary_binop "||"
            | SA.FMatMult -> fmat_mult)
      | SA.SUnop(op, e) ->
          (match op, fst e with
            | SA.INeg, _ | SA.FNeg, _ -> "-"
            | SA.BNot, A.Mat(A.Bool, 1, 1) -> "!"
            | SA.BNot, A.Mat(A.Bool, 1, _) -> "not"
            | SA.Int2Float, _ | SA.Bool2Float, _
            | SA.Float2Int, _ | SA.Bool2Int, _
            | SA.Int2Bool, _ | SA.Float2Bool, _ -> string_of_typ typ
            | _, _ -> raise (Failure "shouldn't get here")) ^ "(" ^ expr 
    env e ^ ")"
      | SA.STypeCons(elist) -> (match typ with
          A.Mat(_, _, _) | A.Array(_, _) ->
            let elist' = expr_list env elist
            in
            string_of_typ typ ^ "(" ^ elist' ^ ")"
        | _ -> raise (Failure ("unexpected type constructor for " ^
                string_of_typ typ)))
      | SA.SNoexpr -> ""

    and expr_list env elist =
      (* handle lists for e.g. function arguments *)
      String.concat ", " (List.map (expr env) elist)
    in

    let stmt env = function
        SA.SAssign(l, r) ->
          let l' = expr env l in
          let r' = expr env r in
          "(" ^ l' ^ ") = (" ^ r' ^ ");\n"
      | SA.SCall(ret, "length", [arr]) ->
          expr env ret ^ " = " ^ "(" ^ expr env arr ^ ").length();\n"
```

```
      | SA.SCall (ret, "sin", [e]) ->
          expr env ret ^ " = " ^ "sin(" ^ "(" ^ expr env e ^ ")" ^ ");\n"
      | SA.SCall (ret, "cos", [e]) ->
          expr env ret ^ " = " ^ "cos(" ^ "(" ^ expr env e ^ ")" ^ ");\n"
      | SA.SCall (ret, "sqrt", [e]) ->
          expr env ret ^ " = " ^ "sqrt(" ^ "(" ^ expr env e ^ ")" ^ ");\n"
       | SA.SCall (ret, "floor", [e]) ->
          expr env ret ^ " = " ^ "floor(" ^ "(" ^ expr env e ^ ")" ^ ");\n"
      | SA.SCall (ret, "pow", [base; power]) ->
          let base' = expr env base in
          let power' = expr env power in
          expr env ret ^ " = " ^ "pow(" ^ "(" ^ base' ^ ") , " ^
                 "(" ^ power' ^ ") );\n"
      | SA.SCall((A.Void, SA.SNoexpr), name, elist) ->
          let elist' = expr_list env elist
          in
           StringMap.find name func_table.scope ^ "(" ^ elist' ^ ");\n"
      | SA.SCall(ret, name, elist) ->
          let elist' = expr_list env elist
          in
          let ret = expr env ret
          in
           ret ^ " = " ^ StringMap.find name func_table.scope ^ "(" ^
    elist' ^
           ");\n"
     | SA.SReturn(e) -> let e' = expr env e in
          if env.cur_qualifier = A.Vertex then
            "gl_Position = " ^ e' ^ ";\nreturn;\n"
          else
            "return " ^ e' ^ ";\n"
      | SA.SIf(predicate, then_stmts, else_stmts) ->
          "if (" ^ expr env predicate ^ ") {\n" ^
            translate_stmts env then_stmts ^
          "} else {\n" ^
            translate_stmts env else_stmts ^
          "}\n"
      | SA.SLoop(body, continue) ->
          let continue' = translate_stmts env continue in
          let env' = { env with forloop_update_statement = continue' } in
          let body' = translate_stmts env' body in
          "while (true) {\n" ^ body' ^ continue' ^ "}\n"
      | SA.SBreak -> "break;\n"
      | SA.SContinue -> env.forloop_update_statement ^ "continue;\n"
    in

    String.concat "" (List.map (stmt env) slist)

  in
```

```
  let add_locals env fdecl =
    List.fold_left (fun (env, locals) bind ->
      let env, local = string_of_bind env bind in
      (env, locals ^ local ^ ";\n")) (env, "") fdecl.SA.slocals
  in

  let string_of_func fdecl =
    let env, formals = List.fold_left (fun (env, formals) (qual, bind) ->
      let env, bind' = string_of_bind env bind in
      let formal = (match qual with
          A.In -> ""
        | A.Out -> "out "
        | A.Inout -> "inout "
        | A.Uniform -> raise (Failure "unreachable")) ^ bind'
      in
      (env, if formals = "" then formal else formals ^ ", " ^ formal))
    (env, "") fdecl.SA.sformals
    in
    let env, locals = add_locals env fdecl
    in
    let env = { env with cur_qualifier = fdecl.SA.sfqual }
    in
    let name = StringMap.find fdecl.SA.sfname func_table.scope
    in
    string_of_typ fdecl.SA.styp ^ " " ^ name ^
    "(" ^ formals ^ ") {\n" ^ locals ^ translate_stmts env fdecl.SA.sbody 
    ^ "}"
  in

  let glsl_funcs = String.concat "\n\n" (List.map string_of_func
    (List.filter (fun fdecl ->
      fdecl.SA.sfqual = A.GpuOnly || fdecl.SA.sfqual = A.Both)
    functions)) ^ "\n\n"
  in

  (* construct a GLSL shader corresponding to a @vertex or @fragment 
    entrypoint
   *)
  let string_of_entrypoint fdecl =
    (* input/output parameters become global in/out variables *)
    let io_decls, _, env = List.fold_left (fun (decls, idx, env) (qual, 
    bind) ->
      let env, bind' = string_of_bind env bind in
      let decl = (if fdecl.SA.sfqual = A.Vertex && qual = A.In then
        (* vertex inputs are linked by location *)
        "layout(location = " ^ string_of_int idx ^ ") "
      else
```

```
␣␣␣␣␣␣␣␣"")␣^
␣␣␣␣␣␣(match␣qual␣with
␣␣␣␣␣␣␣␣␣␣A.In␣->␣"in "
␣␣␣␣␣␣␣␣|␣A.Out␣->␣"out "
␣␣␣␣␣␣␣␣|␣A.Uniform␣->␣"uniform "
␣␣␣␣␣␣␣␣|␣A.Inout␣->
␣␣␣␣␣␣␣␣␣␣␣␣raise␣(Failure␣("inout on entrypoints not supported yet")))␣^
␣␣␣␣␣␣bind'␣^␣";\n"
␣␣␣␣␣␣in
␣␣␣␣␣␣(decls␣^␣decl,␣(if␣qual␣=␣A.In␣then␣idx␣+␣1␣else␣idx),␣env))
␣␣␣␣("",␣0,␣env)␣fdecl.SA.sformals
␣␣␣␣in
␣␣␣␣let␣env,␣locals␣=␣add_locals␣env␣fdecl
␣␣␣␣in
␣␣␣␣let␣env␣=␣{␣env␣with␣cur_qualifier␣=␣fdecl.SA.sfqual␣}
␣␣␣␣in
␣␣␣␣"#version 330\n"␣^
␣␣␣␣"struct dummy_struct {int dummy;};\n\n"
␣␣␣␣^␣glsl_structs␣^␣glsl_funcs␣^␣io_decls␣^
␣␣␣␣(*␣the␣entrypoint␣itself␣becomes␣main()␣*)
␣␣␣␣"\n\nvoid main() {\n"␣^␣locals␣^␣translate_stmts␣env␣fdecl.SA.sbody␣^␣
    "}"
␣␣in

␣␣(*␣return␣a␣map␣from␣entrypoint␣name␣to␣shader␣*)
␣␣List.fold_left␣(fun␣funcs␣fdecl␣->
␣␣␣␣let␣shader␣=␣string_of_entrypoint␣fdecl␣in
␣␣␣␣StringMap.add␣fdecl.SA.sfname␣shader␣funcs)
␣␣StringMap.empty␣(List.filter␣(fun␣fdecl␣->
␣␣␣␣fdecl.SA.sfqual␣=␣A.Vertex␣||␣fdecl.SA.sfqual␣=␣A.Fragment)␣functions)
```

runtime.c

```c
#include <stdio.h>
#include <stdlib.h>
#ifdef __APPLE__
#include <OpenGL/gl3.h> /* Apple, y u special? */
#else
#include <GL/glew.h>
#endif
#include <GLFW/glfw3.h> /* window creation and input handling crap */
#include <stdbool.h> /* for true */
#include <string.h>
#include <assert.h>

struct pipeline {
  GLuint vertex_array;
  GLuint index_buffer;
```

```
  GLuint program;
  GLuint depth_func;
};

GLuint compile_shader(const char *source, GLenum stage)
{
  GLuint shader = glCreateShader(stage);
  glShaderSource(shader, 1, &source, NULL);
  glCompileShader(shader);

  GLint result;
  glGetShaderiv(shader, GL_COMPILE_STATUS, &result);
  if (!result) {
    GLint log_length;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &log_length);
    char *error = malloc(log_length + 1);
    glGetShaderInfoLog(shader, log_length, NULL, error);
    printf("error␣compiling␣shader␣source:\n\n%s\n\n------\n\n%s\n",
        source, error);
    free(error);
    exit(1);
  }

  return shader;
}

void create_pipeline(struct pipeline *p,
                     const char *vshader_source, const char *fshader_source,
                     int enable_depth_test)
{
  // compile shaders
  GLuint vshader = compile_shader(vshader_source, GL_VERTEX_SHADER);
  GLuint fshader = compile_shader(fshader_source, GL_FRAGMENT_SHADER);

  // link shaders
  p->program = glCreateProgram();
  glAttachShader(p->program, vshader);
  glAttachShader(p->program, fshader);
  glLinkProgram(p->program);

  GLint result;
  glGetProgramiv(p->program, GL_LINK_STATUS, &result);
  if (!result) {
    GLint log_length;
    glGetProgramiv(p->program, GL_INFO_LOG_LENGTH, &log_length);
    char *error = malloc(log_length + 1);
    glGetProgramInfoLog(p->program, log_length, NULL, error);
    printf("error␣linking␣vertex␣shader:\n\n%s\n\n-------\nand␣fragment␣
```

```
        shader:\n\n%s\n\n------\n\n%s\n",
            vshader_source, fshader_source, error);
    free(error);
    exit(1);
  }

  glGenVertexArrays(1, &p->vertex_array);
  p->index_buffer = 0;

  if (enable_depth_test)
    p->depth_func = GL_LEQUAL;
  else
    p->depth_func = GL_ALWAYS;
}

GLuint create_buffer(void)
{
  GLuint b;
  glGenBuffers(1, &b);
  return b;
}

/* void *data and size are replaced by an array */
void upload_buffer(GLuint buffer, const void *data, unsigned size, int
    hint)
{
  /* the target doesn't really matter */
  glBindBuffer(GL_ARRAY_BUFFER, buffer);
  glBufferData(GL_ARRAY_BUFFER, size, data, hint);
}

/* location corresponds to an input in the vertex shader */
void pipeline_bind_vertex_buffer(struct pipeline *p, GLuint b, int
    components, int location)
{
  glBindVertexArray(p->vertex_array);
  glBindBuffer(GL_ARRAY_BUFFER, b);
  glVertexAttribPointer(location,
                      components, GL_FLOAT, false, /* vecN - comes from
                          type of buffer */
                      0, /* stride */
                      (void *) 0 /* array buffer offset */
                     );
  glEnableVertexAttribArray(location);
}

GLuint pipeline_get_vertex_buffer(struct pipeline *p, int location)
{
```

```
  GLuint out;
  glBindVertexArray(p->vertex_array);
  glGetVertexAttribIuiv(location, GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING,
      &out);
  return out;
}

int pipeline_get_uniform_location(struct pipeline *p, char *name)
{
  return glGetUniformLocation(p->program, name);
}

void pipeline_set_uniform_float(struct pipeline *p, int location,
                                float *values, int rows, int cols)
{
  glUseProgram(p->program);
  switch (cols) {
    case 1:
      switch (rows) {
        case 1:
          glUniform1fv(location, 1, values);
          break;
        case 2:
          glUniform2fv(location, 1, values);
          break;
        case 3:
          glUniform3fv(location, 1, values);
          break;
        case 4:
          glUniform4fv(location, 1, values);
          break;
        default:
          assert(!"unreachable");
      }
      break;
    case 2:
      switch (rows) {
        case 1:
          glUniform2fv(location, 1, values);
          break;
        case 2:
          glUniformMatrix2fv(location, 1, false, values);
          break;
        case 3:
          glUniformMatrix2x3fv(location, 1, false, values);
          break;
        case 4:
          glUniformMatrix2x4fv(location, 1, false, values);
```

```
        break;
      default:
        assert(!"unreachable");
    }
    break;
  case 3:
    switch (rows) {
      case 1:
        glUniform3fv(location, 1, values);
        break;
      case 2:
        glUniformMatrix3x2fv(location, 1, false, values);
        break;
      case 3:
        glUniformMatrix3fv(location, 1, false, values);
        break;
      case 4:
        glUniformMatrix3x4fv(location, 1, false, values);
        break;
      default:
        assert(!"unreachable");
    }
    break;
  case 4:
    switch (rows) {
      case 1:
        glUniform4fv(location, 1, values);
        break;
      case 2:
        glUniformMatrix4x2fv(location, 1, false, values);
        break;
      case 3:
        glUniformMatrix4x3fv(location, 1, false, values);
        break;
      case 4:
        glUniformMatrix4fv(location, 1, false, values);
        break;
      default:
        assert(!"unreachable");
    }
    break;
  default:
   assert(!"unreachable");
 }
}

void pipeline_set_uniform_int(struct pipeline *p, int location,
                              int *values, int rows, int cols)
```

```c
{
  glUseProgram(p->program);
  switch (cols) {
    case 1:
      switch (rows) {
        case 1:
          glUniform1iv(location, 1, values);
          break;
        case 2:
          glUniform2iv(location, 1, values);
          break;
        case 3:
          glUniform3iv(location, 1, values);
          break;
        case 4:
          glUniform4iv(location, 1, values);
          break;
        default:
          assert(!"unreachable");
      }
      break;
    default:
      assert(!"unreachable");
  }
}

void pipeline_get_uniform_float(struct pipeline *p, int location,
                                float *values)
{
  glUseProgram(p->program);
  glGetUniformfv(p->program, location, values);
}

void pipeline_get_uniform_int(struct pipeline *p, int location,
                              int *values)
{
  glUseProgram(p->program);
  glGetUniformiv(p->program, location, values);
}

void read_pixel(int x, int y, float *pixel)
{
  glReadPixels(x, y, 1, 1, GL_RGBA, GL_FLOAT, pixel);
}

void init(void)
{
  if (!glfwInit()) {
```

```
    fprintf(stderr, "failed␣to␣initialize␣glfw\n");
    exit(1);
  }

  /* OpenGL 3.3, core profile */
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); /* To make MacOS
      happy; should not be needed */
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); /* We
      don't want the old OpenGL */

}

/* TODO: when we get string support, add a string for the name */
GLFWwindow *create_window(int width, int height, int offscreen) {
  if (offscreen)
    glfwWindowHint(GLFW_VISIBLE, false);
  else
    glfwWindowHint(GLFW_VISIBLE, true);

  GLFWwindow *window = glfwCreateWindow(width, height, "Blis", NULL, NULL);
  if (!window) {
    fprintf(stderr, "failed␣to␣create␣window\n");
    exit(1);
  }

  glfwSetInputMode(window, GLFW_STICKY_KEYS, true);
  glfwSetInputMode(window, GLFW_STICKY_MOUSE_BUTTONS, true);

  return window;
}

void set_active_window(GLFWwindow *window)
{
  glfwMakeContextCurrent(window);

#ifndef __APPLE__
  glewExperimental = true;
  if (glewInit() != GLEW_OK) {
    fprintf(stderr, "Failed␣to␣initialize␣GLEW\n");
    exit(1);
  }
#endif

  // we always enable depth test
  glEnable(GL_DEPTH_TEST);
}
```

```c
void clear(float *color)
{
  glClearColor(color[0], color[1], color[2], color[3]);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

void draw_arrays(struct pipeline *p, int num_indices)
{
  glUseProgram(p->program);
  glBindVertexArray(p->vertex_array);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, p->index_buffer);
  glDepthFunc(p->depth_func);
  if (p->index_buffer)
    glDrawElements(GL_TRIANGLES, num_indices, GL_UNSIGNED_INT, (void*)0);
  else
    glDrawArrays(GL_TRIANGLES, 0, num_indices); /* Go! */
}

void swap_buffers(GLFWwindow *window)
{
  glfwSwapBuffers(window);
}

void poll_events(void)
{
  glfwPollEvents();
}

bool should_close(GLFWwindow *window)
{
  return glfwWindowShouldClose(window);
}

struct blis_string {
  int size;
  char *str; // note: NOT NUL-terminated
};

void read_file(struct blis_string *file, struct blis_string path)
{
  char *fpath = malloc(path.size + 1);
  // Out-of-memory error? What error?
  memcpy(fpath, path.str, path.size);
  fpath[path.size] = '\0';

  FILE *f = fopen(fpath, "r");
  if (!f) {
```

```
    fprintf(stderr, "couldn't open file %s\n", fpath);
    exit(1);
  }

  fseek(f, 0, SEEK_END);
  int fsize = ftell(f); // 2^31 ought to be large enough for everyone...
  fseek(f, 0, SEEK_SET);

  file->size = fsize;
  file->str = malloc(fsize);
  fread(file->str, fsize, 1, f);

  fclose(f);
}

void print_string(struct blis_string str)
{
  fwrite(str.str, str.size, 1, stdout);
}
```