Programming Project 4: LLVM IR Generator

1. JIT compiler gli

GLSL JIT compiler, aka *gli*, is adapted from the existing LLVM *lli* execution engine. It serves as a verifier for your compiler. Though we don't provide a reference LLVM BC generator, with gli, at least you can run it by your own to see if your implementation is correct.

2. Matrix and vector

There is a known limitation with *gli* – because *lli* doesn't handle matrices and vectors properly if they are used as function arguments or return values.

So, we will NOT have any test cases which use matrix or vector as return or arguments. But other use cases in the tests are possible:

- Vector can be present as global or local variables, and do regular arithmetic operations (see section 3 below).
- Matrix is NOT required in any test cases (except for extra credits).

3. **Matrix and vector operations**

In P3, we didn't have error reporting functions for mixed operands of matrix / vector with scalar values. But these are indeed valid GLSL programs.

In LLVM IR generator, since we don't need to check for semantic errors (all provided tests are legal GLSL programs), it is possible to handle these types of operations.

Legal operations between scalar, vector, and matrix operands are listed on GLSL spec 5.9 (pages 106/107). Again,

- Vector related operations are required.
- Matrix related operations are for extra credits.

4. **Example of load/store**

Here is an example of GLSL program and possible LLVM IR generation.

```
float a;
float b;
float c;
b = 1.0;
a = b + 2.0;
c = a + b;
```

The load/store based LLVM IR may look like:

```
store float 1.0, float * %b
%1 = load float * %b
%2 = fadd float %1, 2.0
store float %2, float * %a
%3 = load float * %a
%4 = load float * %b
```

```
%5 = fadd float %3, %4
store float %5, float * %c
```

An optimized sequence (without constant folding) may look like:

```
store float 1.0, float * %b
%1 = fadd float 1.0, 2.0
store float %1, float * %a
%2 = fadd float %1, 1.0
store float %5, float * %c
```

A more optimized sequence (with constant folding) may look like:

```
store float 1.0, float * %b
store float 3.0, float * %a
store float 4.0, float * %c
```

These stores may even be eliminated after propagating the constant values.

5. **LLVM classes and API documentation**

In the past, many students found it challenging to get good documents on LLVM APIs. While this remains as a challenge, you may find this site useful:

- http://llvm.org/docs/doxygen/html/classllvm_1_1BinaryOperator.html
-
- From here, you can link to other classes and APIs.