Project 4: LLVM IR Generation

The Goal
In the final project, you will implement an IR generator for your compiler that will generate LLVM IR instructions. Finally, you get to the outcome of all your labor—running GLSL programs!

This pass of your compiler will traverse the abstract syntax tree, stringing together the appropriate LLVM IR instructions for each subtree—to assign a variable, do vector operations, or whatever is needed. Those LLVM instructions are then executed by a modified LLVM JIT compiler without any grungy details of the machine code. Your finished compiler will do IR generation for all valid GLSL programs (i.e. the semantically correct program). Students generally find P4 to be close to P3 in terms of difficulty and time consumption.

Since LLVM 3.4.2 was installed in our lab machines, we will use LLVM version 3.4 as the P4 compiler infrastructure. Note that the latest LLVM releases have changed its IR instruction formats and APIs, you may find some descriptions from llvm.org no longer match what you see on the lab machines. So, if you want to look for the reference manual online, be sure to use this site:

- http://llvm.org/releases/3.4.2/docs/index.html
- http://llvm.org/releases/ (look for 3.4.2)

LLVM has broad debugging utilities. We provide a piece of code with debugging dumps for you to quickly start. The debugging can be intense at times since you may need to drop down and examine the LLVM instructions to sort out the errors. By the time you're done, you'll have a pretty thorough understanding of the LLVM assembly and will even gain a little familiarity with LLVM compiler infrastructure.

Note that we will not implement the actual back end of the GLSL compiler which involves more specific knowledge of GPU architecture. But you still get the awesome feeling when you finally get to the stage where you can compile GLSL programs and execute them.

Starter Files
The starting files are in Piazza resources. The project contains the following files (the boldface entries are the ones you are most likely to modify, although depending on your strategy you may modify others as well):

- `Makefile` builds project
- `main.cc` **main** and some helper functions
- `scanner.h/l` our scanner interface/implementation
- `parser.y` **bison** parser; replace with your own
- **ast.h/.cc** interface/implementation of base AST node class
- **ast_type.h/.cc** interface/implementation of AST type classes
- **ast_decl.h/.cc** interface/implementation of AST declaration classes

- **ast_expr.h/.cc** interface/implementation of AST expression classes
- **ast_stmt.h/.cc** interface/implementation of AST statement classes
- **irgen.h/cc** template implementation of LLVM IR generator and utilities. This file should be heavily customized for your need.
- symbol.h/cc The symbol table (empty) – use your P3 implementation
- `errors.h/.cc` error-reporting class for you to use
- `list.h` simple list template class
- `location.h` utilities for handling locations, **yylloc**/**yyltype**
- `utility.h/.cc` interface/implementation of our provided utility functions
- `public_samples` directory of test input files

Copy the entire directory to your home directory. Use **make** to build the project which writes out an executable binary, ***glc***. The GL compiler (glc) reads input from **stdin**, so you can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% glc < foo.glsl > foo.bc
% gli foo.bc > foo.myoutput
% diff foo.myoutput foo.out
```

The output from *glc* is LLVM BC binary that can be executed via the LLVM JIT compiler ***gli***. The JIT compiler *gli* takes a LLVM BC file as input, and compile it to generate native machine code for final execution. You don't need to modify anything for JIT compilation; it is transparent to you in this project.

Note that gli also has an implicit input (data file) which specifies the entry function name for execution, and values of the entry function arguments (if any) and global variables (if any). So if you write your own test case, you should provide a corresponding data file (with same name as BC file but .dat as extension).

For example, if you have generated a program foo.bc, whose assembly is shown below (use command: *`llvm-dis foo.bc`* to disassemble .bc file):

```
; ModuleID = 'foo.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-
n8:16:32:64-S128"
target triple = "x86_64-redhat-linux-gnu"

@x = global i32 0, align 4
@y = global i32 0, align 4

define i32 @foo(i32 %a) {
  %1 = load i32* @x, align 4
  %2 = load i32* @y, align 4
  %3 = add i32 %2, %1
  %4 = add i32 %3, %a
  ret i32 %4
}
```

The corresponding data file, *foo.dat*, may look like (color coded below):

```
funct: foo
param: int, 11
gin:   x, int, 4
gin:   y, int, 6
```

Here is the description of the data file format, key-vale pairs:

- "**funct:**" – this key defines the GLSL program entry point, "*foo*".
- "**param:**" – this key defines the function parameter, whose type is "*int*", with an integer value of "*11*".
- "**gin:**" – this key declares a global input whose name is "*x*", type "*int*", with an integer value of "*4*".
- "**gin:**" – this key declares a global input whose name is "*y*", type "*int*", with an integer value of "*6*".

You can declare as many "param:" and "gin:" per your test case's needs. There is no particular order among these declarations. Similarly, you may declare multiple "funct:" entry points, but only the first one takes effect.

As always, the first thing to do is to carefully read all the files we give you and make sure you understand the lay of the land. This is particularly important here since there is a chunk of new code in the project. A few notes on how you start the IR generation work:

- You are given the same parse tree node classes as P3. Your job is to add **Emit** functions to the nodes, implemented using the same virtual method business you used for **Print** and **Check** in previous programming projects. You can decide how much of your P3 semantic analysis you want to incorporate into your p4 project. We will not test on GLSL programs with semantic errors, so you are free to disable or remove your semantic checks so as to allow you to concentrate on your new task without the clutter.
- We are using the exactly same GLSL grammar before. We provide **parser.y** which works with our reduced GLSL grammar (reference parser same as P3 except a minor enhancement). You don't need to make changes to the parser or rearrange the grammar, but you can if you like.

Implementing IR Generation

Here is a quick sketch of a reasonable order of attack:

- Before you begin, go over the LLVM instructions and some short examples in the LLVM Language Reference manual to ensure you have a good grasp on BC and LLVM module structure.
- While traversing the AST tree nodes to emit instructions, the root node ("program") can map to LLVM module. In this node, each variable decl can be generated as LLVM GlobalVariable, and each function decl as a LLVM function.
    - Since the output .bc file will be executed on the native machine, it is critical to specify the LLVM module with correct target triple and data layout.

- The test is based on ieng6 machines. We have pre-defined the machine target as below (see irgen.cc). When you customize the codegen functions we provide, please make sure you don't change any of these strings.

```
const char *IRGenerator::TargetLayout = "e-p:64:64:64-i1:8:8-i8:8:8-
i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-
a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128";

const char *IRGenerator::TargetTriple = "x86_64-redhat-linux-gnu";
```

- Within a function body, there may be many local variables. Local variables are generated as *alloca* in LLVM IR. Typically, LLVM requires all *alloca* instructions at the first basic block of the function body. So, it is a good idea to always create an entry block in the function body. Thus you can insert *alloca* instructions here without worrying about interleaving other instructions with *alloca*.
- There are different strategies to generate LLVM IR, based on pure SSA or load/store based generation. For simplicity, you may just use load/store based generation so that most of time you only need to scan AST tree once. We will provide a simple example of loads/stores in project extra description file.
  - But if you think you can take on one of the optimization ideas for extra credits (see Grading section), feel free to do so.
- Generating code for the control structures (**if**/**while**/**for**) will teach you about labels and branches. You will need to keep the basic blocks for these constructs in lexical order to make sure the LLVM structure is well formed. For example, always generate IF-THEN-ELSE-ENDIF blocks in the order.
- LLVM IR doesn't have swizzles, so, you must simulate it via *shufflevector* instruction.
- Matrix is represented as an array of vectors in LLVM IR. You can use *getelementptr* instruction to access one entire column or one element within a particular column. But we will skip matrix type in this assignment.
- Boolean is represented as *i1* in LLVM IR, where "true" is evaluated to 1, and "false" 0.

Note that it is **not** expected that you generate the LLVM instruction sequence which exactly matches ours. Depending on your strategy, you can get many functionally equivalent results from different sequences. So we will not use *diff* on the LLVM assembly sequences; instead, we diff the ***gli*** output.

- But it is definitely a good idea to disassemble .bc and visually examine if there is a problem whenever you find a mismatch in *gli* output.

The debug flag can be set with ***-d ast*** when invoking the program or programmatically via **SetDebugForKey**. This is quite useful when you are developing the code generator. Most of time, ast semantics may be mapped directly to LLVM IR.

We included comments in the header files to give an overview of the functionality in our provided classes but if you find that you need to know more details, don't be shy about

opening up the **.cc** file and reading through the implementation to figure it out. You can learn a lot by just tracing through the code, but if you can't seem to make sense of it on your own, you can send us email or come to office hours.

Testing
There are various test files that are provided for your testing pleasure in the samples directory. For each GLSL test file, we also have provided the data file as an implicit input when executing that program's bitcode under LLVM JIT compiler **gli**. There are many different correct ways of generating the instructions, so it's not important to compare LLVM BC disassembly, but the runtime output should match.

Be sure to test your program thoroughly, which will certainly involving making up your own additional tests. We will test your compiler only on syntactically and semantically valid input. We won't expect your final submission to report any errors.

- When developing your own tests, make sure your GLSL program is free of syntax and semantic errors!

Grading
The final submission is worth 20% of your overall grade in the course. We will thoroughly test your submission against many samples. We will run LLVM bitcode produced by your compiler on the LLVM JIT compiler **gli** and **diff** against the correct runtime output.

There are many opportunities for extra credit on this assignment, and we'd be more than happy to reward you for going above and beyond what's required. Here are a couple of cool ideas:

- **Add optimization**. Your generated code needn't be efficient for this assignment, but that doesn't mean that you can't try to optimize your generated IR. Any optimizations that you do, provided that they are correct, will earn extra credit, as long as you document it in your README.
- **Matrix.** Matrix operations, such as mat * vec, or mat * scalar, matrix assignment are good examples.

Good luck!