CSC467 lab2

Yiming Kang 998676730
TianYu Li 999107149

Please see my github repo for the commit history git@github.com:yimingkang/CSC467_hw2

Approach:

The first step we took was to code in the gramma rules without any modification or output and see what happens. Bugs were discovered and fixed and tests were added.

1. Type in the gramma, then `make`
2. Got shift/reduce and reduce/reduce conflict, fixed the conflicts
3. Added yTRACE() call when each rule is reduced, this is done via a script `gen\_yTRACE.py`. `yTRACE()` prints the exact rule that is being reduced at the moment. This allows us to visualize the tree being constructed so that we can verify the correctness of the tree for a particular synatx.
4. Improved testing framework (baesd on the one we built for lab1)

Challenges:

These are the challenges we found noteworthy (among others):

1. Dangling else shift/reduce conflict. We decided to give statements with 'else' a higher priority (shift rather than reduce) which is Bison's default
2. Figuring out `%prec SYMBOL` assigns the given rule the priority of `SYMBOL` - this is necessary to resolve some of the shift/reduce conflicts
3. Systematic testing: Since we're implementing parser.y, we cannot actually judge whether an **output** is correct. We can only feed it with files known to be syntactically correct and expect it to pass, and files known to be syntactically invalid to fail. Particularly, we need to test the order of precedence given by miniGLSL specification is handled correctly by our parser.

Testing:

As mentioned above, we have no good way of knowing if the actual y.output is correct because we defined it. We were however, able to come up with a more elaborate testing framework than simply checking whether a file passes or failes.

Test location: All tests are located in `./tests`. Tests for lab #x are located in `./tests/labx_test`
Running tests: Run `make && make test`. This will first invoke `python gen_yTRACE.py parser.y.bak > parser.y` to generate the actual rules, then compile and run tests

1. Building upon the testing framework of lab1, we write the test code for the parser to recongize in each .frag file. The expected the output for each .frag file is the .target file by the same name. We use a target.sh to automatically generate the expected output file and manually verify them. We verfy the correctness of later compiler output by running `diff` on output and target. The bison parser is a bottom up parser, hence the order of each rule recognition tells us about the synatex tree generated. We look at each rule being recongized and verify that it is what we expect. Please look at .target for the expected parse output of each line
2. As mentioned in challenges, we tested whether files known to be correct actually passes and files known to be incorrect actually fails.

3. The test cases cover all the valid synatex of miniGLSL language. We look at the output of the parser to confirm that the synatex tree generated for each line is what we expect.

Test cases:

1. lab2_test/general_test.frag
   this test case contains all the valid synatx of miniGLSL. Comments in the test case explains what each line of code is testing for the test case tests the following. We look for the synatex generated for each line of code and it should agree with the grammer defined for miniGLSL
   - declarations, constructors, argument list
   - expressions with operators (+,-,/,*, ^,(),)
   - IF ELSE statment, WHILE statement and nested WHILE statement
   - expressions with compare operators
   - vector indexing
   - function, argument lists
   - scope

2. lab2_test/order_of_precedence.frag This test case test all the combinations of operators (*,-,+,/,&&,||,(),>,<,==,!=, <=,>=) in an expression and that the order of precedence of each is handled correctly. The dangling else problem is also tested here. Comments in this test case shows what each line is testing for.

3. lab2_test/shader.frag and lab2_test/phong.frag These are the sample programs provided. We compile them with -Tp flag and make sure they pass