Iteration 4: C++ code generation.

Updates

Updates, Corrections, etc will be posted here - please check for these daily.

- UPDATE

  ### April 18:

  Replaced `matrixRead` with `matrix_read` in all sample and .cc translation files in which they were used. Replaced `numRows` and `numCols` with `n_rows` and `n_cols` in all sample and .cc translation files in which they were used. Updated `Matrix.h` to use `matrix_read` , `n_cols` , and `n_rows` . Updated `codegeneration_tests.h` to have a space after the `-I.` , and to use the scanner namespace.

- UPDATE

  ### April 17:

  We updated each of the .cc files generated from the sample_1.dsl, sample_2.dsl, sample_3.dsl, sample_7.dsl, and sample_8.dsl files to have the directive `#include "include/Matrix.h"` We also changed the name of the method `readMaxtrix` in `Matrix.h` to `read_matrix` We also updated the codegeneration_tests.h file by changing `readInput` to `read_input`, and included the: `-I.` option in the command to compile the translated files on line 59

- **Due Date: All required work must be submitted by *11:55pm on Friday, April 28* .**
- **Late work will *penalized in cumulative increments of 5 percent per day late* for this final iteration. Thus, if you submit your work 1 day late, you incur a 5 percent penalty, submitting 2 days late incurs a 10 percent penalty and so forth.**
- ***Submissions after Friday, 5/5 at 11:55pm will be assigned a zero and neither re-grade work for iteration 3 or iteration 4 will be carried out.***

OPTION: Work alone or continue to work in your team.

In this iteration you can continue working in your current team or you can work individually.

If you decide to work individually you need to copy files into your individual repository. Run the following command from your "project" directory in your group repository.

```
git archive --format zip --output ~/project.zip master
```

This will create a "project.zip" file in your home directory. You can then "unzip" this file and move the appropriate files into your individual repository. These files are now free of the ".git" files that exist in your working copy.

If you decide to work alone, be sure to tell your partner; your discussion TA, and send an email to csci3081-help@cs.umn.edu telling us that you will be working on iteration 4 alone.

Iteration 4 Tasks

A new method for AST classes.

For this final iteration you will extend your AST classes to translate a given program written in the Forest Cover Analysis Language to C++. There is one primary step (and some secondary ones discussed later) that you need to take to implement this.

1. You need to implement the following method for the appropriate AST classes:

```
std::string CppCode ( ) ;
```

When called from the root node of an AST, this method should return the complete C++ program that implements the dsl program that was input to the translator.

Getting Started

We will discuss this iteration in detail, and continue to fill out the full specification for this iteration and discuss new tests for assessing this iteration in upcoming lab and class sessions.

To get started, start with the sample versions of the sample programs - which can be found in the files `sample_1.dsl` and `sample_2.dsl` . These files and the additional dsl test files, along with and their C++ translations from our solution in are in the Git class repository (class-repo) in the ProjectFiles/Iter_4_Files/samples directory Attempting to implement the code necessary to generate C++ code for these sample dsl programs first is a good way to start this iteration. We will discuss some of the issues with matrix types in upcoming lab and class sessions

Note that the sample translations to C++ can vary and will change to include a matrix library header file.

Some test programs, their translation, sample data, and expected output.

The ProjectFiles/Iter_4_Files/samples directory contains several sample Forest Cover Analysis Language programs, their C++ translation generated by our solution, sample data, and their expected output.

The Forest Cover Analysis Language programs have a ".dsl" (domain specific language) extension, the C++ translations end in "_trans.cpp", the expected output for each has a ".expected" extension.

The tests in "codegeneration_tests.h" (see below) looks for these files in the "samples" directory - thus, if you should copy all of the files located in the class repository (class-repo)in the ProjectFiles/Iter_4_Files/samples directory into your "samples" directory and add them to your git repository. Note, use the updated files in your iteration 4 samples directory for this iteration

- sample_1.dsl - print statements
- sample_2.dsl - repeat and print
- sample_3.dsl - let expression
- sample_7.dsl - matrix operations
- sample_8.dsl - matrix read and print
- forest_loss_v2.dsl

Below are the contents of the file: "codegeneration_tests.h"

```
#include <cxxtest/TestSuite.h>
#include <iostream>
#include "include/parser.h"
#include "include/read_input.h"

#include <stdlib.h>
#include <string.h>
#include <cstring>
#include <fstream>

using namespace std;
using namespace fcal;
using namespace parser;
using namespace ast;

class CodeGenTestSuite : public CxxTest::TestSuite
{
public:
```

```
    Parser p ;
    ParseResult pr1 ;


    void writeFile ( const string text, const string filename ) {
        ofstream out(filename.c_str());
        out << (string) text << endl ;
    }

    void codegen_tests ( string filebase, bool checkExpected ) {
        string file = filebase + ".dsl" ;
        string path = "./samples/" + file ;
        string cppbase =  "./samples/" + filebase ;
        string cppfile =  cppbase + ".cc" ;
        string cppexec =  cppbase ;
        string cppout = cppbase + ".output" ;
        string expected = cppbase + ".expected" ;
        string diffout = cppbase + ".diff" ;

        int rc = 0 ;

        // 1. Test that the file can be parsed.
        ParseResult pr1 = p.Parse(ReadInputFromFile( path.c_str()));
        TSM_ASSERT(file + " failed to parse.", pr1.ok()) ;

        // 2. Verify that the ast field is not null
        TSM_ASSERT(file + " failed to generate an AST.", pr1.ast() != NULL);


        // 3. Verify that the C++ code is non-empty.
        string cpp1 = pr1.ast()->CppCode() ;
        TSM_ASSERT ( file + " failed to generate non-empty C++ code.",
                     cpp1.length() > 0 ) ;

        writeFile ( cpp1, cppfile ) ;

        // 4. Compile generated C++ file
        string compile = "g++ ./src/Matrix.cc -I. " + cppfile +
                         " -o " + cppexec ;
        rc = system ( compile.c_str() ) ;
        TSM_ASSERT_EQUALS ( "translation of " + file +
                            " failed to compile.", rc, 0 ) ;

        string cleanup = "rm -f " + cppout ;
        system ( cleanup.c_str() ) ;

        // 5. Run the generated code.
        string run = cppexec + " > " + cppout ;
        rc = system ( run.c_str() ) ;
        TSM_ASSERT_EQUALS ( "translation of " + file +
                            " executed without errors.", rc, 0 ) ;

        // 6. Check for correct output.
        if ( checkExpected ) {
            string diff = "diff " + cppout + " " + expected + " > " + diffout ;
            rc = system ( diff.c_str() ) ;
            TSM_ASSERT_EQUALS ( "translation of " + file +
                                " did not produce expected output.", rc, 0 ) ;
        }
    }

    void test_sample_1 ( void ) { codegen_tests ( "sample_1", true ); }
    void test_sample_2 ( void ) { codegen_tests ( "sample_2", true ); }
    void test_sample_3 ( void ) { codegen_tests ( "sample_3", true ); }
    void test_sample_7 ( void ) { codegen_tests ( "sample_7", true ); }
    void test_sample_8 ( void ) { codegen_tests ( "sample_8", true ); }


    /* You should create .expected files in ../samples for these with the expected
     * output of the two FCAL programs you create. You can then change the second argument to true to
     * validate them after you get the translation working . */
    void test_your_code_1 ( void ) { codegen_tests ( "my_code_1", false ) ; }
    void test_your_code_2 ( void ) { codegen_tests ( "my_code_2", false ) ; }

    void test_forest_loss ( void ) { codegen_tests ( "forest_loss_v2", true ); }
} ;
```

You should note that the C++ code that you generate does not have to match the translations that we have provided for you. The primary reason to show you these translations is to help with questions regarding the behavior of Forest Cover Analysis program constructs. Also, the layout (white-space) of your C++ programs is completely up to you. You translator should generate C++ code that is at least somewhat readable.

You will also see a **#include "include/Matrix.h"** line in the generated C++ files. Part of this iteration, and lab 12 (which will

occur Friday, April 14), will be to design and implement this Matrix code. Guidance can be found by looking at the generated C++ code, and upcoming discussions in lecture, lab 12, and lab 13 will help to provide more detailed specifications.

Write two "interesting" programs in the Forest Cover Analysis Language (FCAL)

You also need to write two "interesting" sample programs in the Forest Cover Analysis Language and include them in your testing. One of the test programs should implement matrix multiplication for 2 matrices whose values are obtained from input files that you create, and a while loop should be used in at least one of the test programs you formulate. In both cases, your test programs should compute a value or set of values and produce output. Thus, you will have to create `.expected` files for each of the test programs you write, and those files should contain the expected output for each of the test programs that you write.

Name these **"my_code_1.dsl"** and **"my_code_2.dsl"** and put them in your "samples" directory. You will want to replace the contents of these files with your own interesting programs written in the FCAL Lanugage. You will also need to include at least one ".dat" (data) file for the test program that reads in the matrices.

As part of this iteration, you should overload the assignment operator and multiplication operator for matrices, and successfully use them in one or both of the dsl program you are required to write. For example, you can use them to multiply matrices together. So for example, given the matrices

```
m1 =  1 2
      3 4
```

and

```
m2 = 4 3
     2 1
```

```
The contents of m3, after the statement
```

```
m3 = m1 * m2;
```

```
is executed are:
```

```
8 5
20 13
```

Note, the overloaded * operator should work for multiplying both square matrices (n x n) and non-square matrices (n * m) - and should print an error if two non-square matrices (for example a 3 x 4 matrix and a 5 x 3 matrix) cannot be multiplied together.

Also, the assignment operator should only work for matrices with the same dimensions. Thus, for example, if an attempt to assign a two-row by two column matrix values from a two row by 3 column matrix is made, an error should be printed.

Testing

A test suite is provided in the file codegeneration_tests.h located in ProjectFiles/Iter_4_Files/src directory

It tests the sample programs above, the two FCAL .dsl programs you are required to create, and the forest loss program.

You should add a target to your make file, and ensure that your makefile will create an executable file named "codegeneration_tests" based on the tests in codegeneration_tests.h.

In our solution, all of the AST code necessary to translate the FCAL programs in .dsl files to C++ programs in .cc exists in two files - `AST.h` and `AST.cc` . The code for the matrix class is implemented in the files `Matrix.h` and `Matrix.cc`.

Assessment

In assessing your code we will copy our original version of the "codegeneration_tests.h" file over the one in your directory and then run the command "codegeneration_tests" from the command line after running "make codegeneration_tests".

One of the first things you should do is make sure that the tests run to completion without any segmentation faults. Many of them will initially fail, but you can then start working.

You may, for example, comment out all the calls to codegen_tests except the first one and then focus on implementing what you need to implement in order to get `sample_1.dsl` to pass all of the tests. Then proceed by working through the rest of the programs. Better yet, start by creating a very simple program as a test program (as we did in iteration 3), get that working and proceed from there.

What to turn in

Your code must be committed to your individual or group git project repository in and be properly tagged as "iter4". This should have the same structure as it did in the previous 3 iterations.

If continue working in a team, you are also required to document how you and your partner(s) split up the work for this iteration. You must specify what parts of the design each of you did, or if this work was done jointly. For design work you are strongly encouraged to work together. You must also specify what parts of the code each of you have completed - some of this will be work done individually, other parts may be done jointly, perhaps using pair programming techniques. Finally, you should include a log of your commits. This document should be part of your "iter4" tagged files and reside in a plain text file named **iteration4_work.txt**

Regrading past iterations: If your solution now passes tests that it did not previously pass on iteration 3 then you may ask that we re-grade those those test cases to award some of the points that you lost during the original assessment. **You must email csci3081-help@cs.umn.edu to request the regrading - if you do not then your original grades on iteration 3 will be kept.**

Grading Criteria

```
Possible points Iteration 4: 150.

Code Development Process: 50 points.
-----------------------------------
Division of work statement (10 points).
___ /10: A non-trivial, well-written, meaningful, truthful, well-formatted,
        spell-checked and gramatically-correct division of work statement.
                Include your commit log and comments about it.


___/ 15: Git is used properly - e.g, meaningful comments with each commit and frequent commits by
different team members;
                 Code adheres to the google style guidelines: files are in their proper
                 directories; code in .cc and .h files passes the cpplint.py style checker
                 and conforms to the following google style guidelines (unless we specify otherwise):

        -Use of C++, not C-style casts
        -No use of C-style memory allocation/copy operations
        -Usage of namespaces, including proper naming
        -Initialization of variables when they are declared
        -All 1 parameter constructors are marked explicit
        -All inheritance passes the 'is-a' test
        -All data members are private in each class
        -Proper ordering of declarations within a class (functions->data; public->protected->private)
        -All references passed to functions are labelled const.
        -Usage of nullptr, rather than NULL
        -Proper file naming
        -Proper function naming (use your discretion for "cheap")
        -Proper data member naming
        -Minimization of work performed in constructors
        -Not using exceptions
        -Proper commenting throughout the header files and source files

                 Note: Files in the tests and samples directories are exempt from google style
requirements.




Successful compilation and generation of tests cases.  (15 points)
Each of the following commands succeeds with no errors.
___ / 5: Legacy tests from iteration 1, iteration 2, and iteration 3 pass without errors
___ / 5: make codegeneration_tests.cc
___ / 5: make codegeneration_tests


Running ALL test cases doesn't result in a segmentation fault. (10 points)
Test cases may fail, but should not cause the testing framework to
crash.
```

___ / 10:

Passing Test Cases - the code generation tests in the file:

codegeneration_tests.h : 100 points.

___ / 10: passes the test: test_sample_1

___ / 10: passes the test: test_sample_2

___ / 10: passes the test: test_sample_3

___ / 10: passes the test: test_sample_7

___ / 15: passes the test: test_sample_8

___ / 15: passes the test: test_my_code_1
(test_my_code_1 implements matrix multiplication by reading
in two matrices from two input files, and then multiplies
them together using the overloaded matrix multiplication
and assignment operators)

___ / 15: passes the test: test_my_code_2
(test_my_code_2 should use a while loop to compute
some interesting value or set of values)

___ / 15: passes the test: test_forest_loss_v2