Iteration 3 - Designing and Implementing Abstract Syntax Trees (ASTs). (ASTs will be used to facilitate the translation of FCAL programs to C/C++ programs in iteration 4)

Posted March 9, 2017

Updates:

Update: March 20th: Note, the due dates for Iteration 3 have changed- review them carefully. The requirements for the writing assignment 1 have also changed - review them carefully

There are THREE (3) assignments for Iteration 3, Two Writing Assignments and One Project Assignment:

1. **Writing Assignment 2: Designing the AST class hierarchy using UML. (50 Points)**
2. **Project Assignment 3: Implementing the design of your class hierarchy and making the changes necessary to your Iteration 2 solution so your translator can successfully construct ASTs. (125 Points)**
3. **Writing Assginment 3: Documenting your code in a dOxygen compliant fashion according to the google stylpe guidelines posted on the class moodle site and the specifications in the rubric below (50 Points)**

Assignment Due Dates:

1. ***The First Version of the UML Design (pdf file) is Due at the start of class, on paper, Thursday, March 30th***
2. ***The Final Version of the UML Design Document (pdf file)due at Tuesday, April 11 at 11:55pm***
3. ***The Project assignment is due Tuesday, April 11, at 11:55pm***
4. ***The dOxygen compliant comments are due with the final version of iteration 3, on Tuesday, April 11, at 11:55pm***

Introduction

For this iteration, you will design a set of C++ classes and augment the parser from iteration to build Abstract Syntax Trees that represent programs written in the Forest Cover Analysis Language (FCAL) - our domain specific language (dsl) designed to detect changes in Forest Cover. Objects of the C++ classes that you design and implement will be the are the components (e.g., nodes)of the abstract syntax trees for FCAL programs.

A set of classes for abstract syntax trees

Much of the design of the classes for this iteration is left to your discretion. You will want to design classes that can be easily extended or modified to accommodate the additional processing that will take place in iteration 4. In iteration 4, you will design and write the code necessary to generate C++ code that implements the code specified in FCAL programs.

For this iteration, you will most likely want to define abstract classes for each "nonterminal" in the grammar described in the comments of parser.cc. Furthermore you may want to define non-abstract sub-classes of these such that each sub-class corresponds to a "production" in that grammar.

You are required to have a super-class to all of these, named `Node`, that is abstract and indicates what methods and fields are required on all objects that are used in that abstract syntax trees.

In upcoming lectures and labs we will sketch out some classes that you may want to fill out and use. For this iteration you are strongly encouraged begin by implementing a small subset of the language and getting that to work. Then move on to add more and more code to cover the rest of the language constructs until you've covered the entire language. One possibility is to first work to parse and unparse a program in the Forest Cover Analysis domain specific language with a single declaration, such as "int x;" - Getting just this to work will be a big step towards the solution **(But you will still have a significant amount of work to do to complete the iteration at that point)**

Part of the assessment of this iteration will be to look for evidence in your work that you have used some of the "good design" techniques that we've discussed, and will discuss, in class. This includes the use of abstract classes and (pure) virtual methods where appropriate.

Specific requirements

To assess your implementation of this iteration we will want to run a standard set of tests as specified in the file

```
ast_tests.h
```

. This imposes a few strict requirements on your implementation and you must satisfy these. This test is shown below:

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

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

using namespace std ;
using namespace fcal::scanner;
using namespace fcal::parser;
using namespace fcal::ast;

class AstTestSuite : public CxxTest::TestSuite
{
public:

    Parser p ;

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

    void unparse_tests ( string file ) {
        string path = "./samples/" + file ;

        // 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 "unparsing" is non-empty.
        string up1 = pr1.ast->unparse() ;
        writeFile(up1,(path+"up1").c_str());
        TSM_ASSERT ( file + " failed to generate non-empty unparsing.",
                     up1.length() > 0 ) ;

        // 4. Verify that the un-parsed string can be parsed.
        ParseResult pr2 = p.parse ( up1.c_str() ) ;
        TSM_ASSERT ( file + " failed to parse the first un-parsing.",
                     pr2.ok ) ;

        // 5. Verify that the ast field is not null after first unparsing
        TSM_ASSERT ( file + " first unparsing failed to generate an AST.",
                     pr2.ast != NULL );

        // 6. Verify that this second unparsing can be parsed.
        string up2 = pr2.ast->unparse() ;
        writeFile(up2,(path+"up2").c_str());
        ParseResult pr3 = p.parse ( up2.c_str() ) ;
        TSM_ASSERT ( file + " failed to unparse the second un-parsing.",
                     pr3.ok ) ;
        string up3 = pr3.ast->unparse() ;
        writeFile(up3,(path+"up3").c_str());
        // 7. Verify that the first and second unparsings are the same.
        TSM_ASSERT_EQUALS ( file + " unparse-1 != unparse-2.", up1, up2 ) ;
        // 8. Verifty that the second and third unparsings are the same.
        TSM_ASSERT_EQUALS ( file + " unparse-2 != unparse-3.", up2, up3 ) ;
    }

    void test_sample_1 ( void ) { unparse_tests ( "sample_1.dsl" ); }
    void test_sample_2 ( void ) { unparse_tests ( "sample_2.dsl" ); }
    void test_sample_3 ( void ) { unparse_tests ( "sample_3.dsl" ); }
    void test_sample_4 ( void ) { unparse_tests ( "sample_4.dsl" ); }
    void test_sample_5 ( void ) { unparse_tests ( "sample_5.dsl" ); }
    void test_mysample ( void ) { unparse_tests ( "mysample.dsl" ); }
    void test_forest_loss ( void ) { unparse_tests ( "forest_loss_v2.dsl" ); }
} ;
```

The file above:

```
ast_tests.h
```

is also in the class_repo Git repository in the tests directory:

```
ProjectFiles/Iter_3_Files/tests
```

This test is relatively simple but imposes some simple and some subtle requirements on your implementation. You are encouraged to think about what these tests really do and what they require before you do too much implementation.

Note the "ParseResult" class has a field name "ast". This is a pointer to an abstract syntax tree (AST). In the parser, when "parseExpr" is called and returns a "ParseResult", this field is a pointer to the AST for the parsed expression. When "parseDecl" is called and returns a "ParseResult", this field is a pointer to the AST of the parsed declaration. When a "Program" is parsed, this field points to the AST for the entire program. Since these trees have root nodes that are of different types, we needed to add a new super-class called "Node" that is a super-class (either directly or indirectly) of the actual types of the created ASTs. In the sample parser snippets given out to you will see this pattern being implemented on the concrete "Root" class (which represents the entire program) and the other classes we share.

This class, as most other classes in the AST class hierarchy, has a method called "unparse" that takes no parameters and returns the un-parsed version of the tree. This is a string (not a "char *" but a C++ "string" datatype). The string is a valid program from our domain specific language and is semantically equivalent to the input program.

As described above, you should write some very simple programs that use a minimum number of constructs from the language so that you can run the test above simple programs. This way you can implement the classes and methods for just a few language constructs and see that things work for a small example. Then as you complete more and more of the implementation you can add these constructs to some sample programs and test them. **The sooner that you can get this test function to pass for a very simple program the better.** Once it works for a very simple program you can the proceed to add more code to cover more constructs in the language.

You might also note that the "c_str" method on the "string" class converts a C++ style string back to a C style string represented as a "char *".

Modifying the parser.cc

A significant challenge, and significant portion of the work in this iteration, is modifying the parser.cc code to fill in the value of the "ast" field. In the parser given to you in Iteration 2, the parseExpr and parseDecl methods (as well as the others) return a ParseResult object. You must modify these methods so that the "ast" field in the ParseResult object returned by the parser is also properly defined.

Parts of the upcoming labs will be devoted to discussing this.

Modifying your Makefile

You will also need to modify your Makefile so that the tests in the file `ast_tests.h` are created and run after the regression tests for the scanner(scanner_tests.h) and parser (parser_test.h) when the `run-tests` target is used. It should also be set up so that the

```
ast_tests
```

executable can be generated, from a Makefile target with this same name, so that the ast_tests can be run separately.

Writing Assignment 2: UML Design of iteration 3

As, part of the process of developing your solution to interation 3, you will develop a UML class design for this iteration. Thus, along with your final deliverable, you will turn in a (collection of) UML class diagrams that show the classes and their relationships, attributes, associations,etc. as they are implemented in your iteration. These can be either computer generated, using a tool like Visio or drawio, - or can be hand-drawn. The primary requirements are that they faithfully describe the classes in your implementation and that they are easy to read and understand. Hand-drawn diagrams that are not precise and clear will not receive full credit. Besides being a tool to develop your design of iteration 3,another key reason for creating these kind of diagrams is to quickly and easily communicate your design to other programmers. Hard-to-read poorly-drawn diagrams do not achieve this objective.

Specifically, you need to create diagrams for at all of your abstract classes and enough of your concrete classes to represent your design

Your collection should include your abstract classes and at least one concrete subclass of each of these.

For example, showing the concrete classes for the assignment statement, if-then/if-then-else statement(s) , while statement and repeat-statement would is appropriate. For expressions, it is redundant to show different classes for all of the infix binary operations (+,-,/,*), provided your implementation has such classes. Showing the design of one of the concrete classes for operations such as these is appropriate

Your UML diagrams must be consistent with your iteration 3 implementation.

You may draw these diagrams by hand or use software to create them. In either case the design needs to be easy to read and organized to minimize the number of arrows that must travel a relatively long distance to reach their target. These should be turned in electronically with your iteration by on Tuesday, April 11th. at 11:55pm. The final UML design that you turn in will be a revised version from an initial version that you develop for the design of iteration 3. You will have to develop an initial design of your iteration 3 solution ON PAPER before the final version, and turn that in by the start of class on Thursday, March 30th. Your initial UML design for iteration 3 will be part of the Lab 9 discussion on Friday, March 24th

Note: Failure to Turn in a paper copy at the start of class on March 30th will result in a 50% penalty for writing assignment 2 (you will lose 1/2 of the 50 points).

Writing assignment 3 for iteration 3

In addition to writing assignment 2 - UML Diagrams, specified below you will be expected to document your code in a dOxygen compliant manner. This will be discussed in Lab 10 on November 13, and is described below

You are required to document your implementation. You should document your code according to the standards in the moodle style guide found on the class moodle page and using the style guide for using dOxygen found here: http://www.yolinux.com/TUTORIALS/LinuxTutorialC++CodingStyle.html

Properties of your comments - which should be present in every .h file and every .cc file that you modify or create

- dOxygen compliant (Compliant with Google style guide on class moodle page and website above for more information on dOxygen)
- correct grammar, spelling or punctuation
- comments are properly indented, google style compliant, dOxygen compliant, and formatted to make them easy to read
- comments are included in the file ast_tests.h that indicate the order in which the test cases were implemented and when they passed.

Finally, comment any additional test cases that you formulate and use, and include them in your tagged deliverable

What to turn in

Your code must be committed to your group GIT repository and tagged as iter3. This should have the same structure as it did in the previous 2 iterations. A pdf file of your final UML design of iteration 3 must be turned in at the same time you submit iteration 3, and it should be placed in the `src` directory of your group GIT repository.

You are also required to document how you and your partner had split up the work of 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 must 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, and should be done using pair programming techniques. This document should be part of your "Iteration3" tagged submission and reside in a plain text file named "iteration3_work.txt" that must be placed inside the "src" directory.

NOTE: It is important that you get started right away as this is a non-trivial assignment and additional writing assignments related to the development of this iteration will be given out in the very near future - and QUIZ 2 will be administered before iteration 3 is due (check your class schedule for details).

Assessment.

```
Possible points for Project Iteration 3 (programming assignment): 125.
Writing Assignment 2 - Developing the UML Design: 50 points
Writing Assignment 3 - Documenting your solution to iteration 3: 50 points

Code Development Process: 50 points.
------------------------------------
(These points are assessed once, on the due date.)

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.


___/ 20: 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.  (10 points)
Each of the following commands succeeds with no errors.

___ / 5: make ast_tests.cc
___ / 5: make ast_tests

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

___ / 10: make run-tests does not result in a segmentation fault.

Passing Test Cases - the unparse tests in the file ast_tests.h: 75 points.
------------------------------
(These points are assessed at each iteration and thus if you fail to
```

pass some test cases now you can get those points back on a later iteration.) Note, that in order to pass the unparse tests, your AST implementation must create an abstract syntax tree that represents the entire test program and produce a string that is equivalent to the entire test program - otherwise you will not recieve credit.

___ / 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_4

___ / 10: passes the test: test_sample_5

___ / 10: passes the test: test_mysample

___ / 15: passes the test: test_forestLossV2

Writing Assignment 2: UML Diagrams for Iteration 3 AST Classes
possible points: 50

___ / 10 Diagrams are easy to read, neatly drawn (or printed) and organized.

____/ 10 UML diagram includes all abstract classes and capture a representative set of concrete classes for each abstract class

____/ 10 UML concrete classes capture the diversity of the classes (if-then-else, assignment, for loop, repeat loop, let, Matrix, etc.).

____/ 10 Correct UML syntax (proper arrows, etc, as described in UML Distilled by Fowler in Chapters 3 and 5 are used).

____/ 10 UML must match your implementation.

Writing Assignment 3: Documentation of implementation of AST classes and methods, their use, and Iteration 3 tests.
Possible points: 50

____/ 30 Google style and dOxygen compliant comments in all of the .h and .cc files that you manually modify or create. Comments in ast_tests.h are dOxygen compliant (test files do not have to adhere to the google style guide).
This means dOxygen can be used to generate documents containing the comments for the .cc and .h files you modify or create.

____/ 10 Correct grammar, spelling, and punctuation should be used in your comments. Also, comments in ast.cc, ast.h and parser.cc, and ast_tests.h should be clear, concise, and accurate (that is, the comments should match the code).

____/ 10 Add comments to the ast_tests.h file that indicate the order and dates when the code necessary to pass the test cases was implemented, and the date that the test passed.