Iteration 2 - Updating the Scanner and integrating it with the Parser (Issued Feb 28, 2017)

Updates:

- Updated March 5th: Made the following updates to fix defects on March 3rd:
    1. Compiler warnings about -Wreorder and -Wreturn-type. This did not cause any erroneous behavior, but was still an error strictly speaking.
    2. ext_token.cc now includes assert.h
    3. The parser was not correctly returning what was it expecting when it encountered an error (i.e. "Expected but found variableName"). This is now fixed and should help with debugging.
    4. Note, if you have not defined a token.h file with your token class, then remove “#include/token.h” directives from the ext_token.cc and parser.cc files. However, we are advising you to define your token class in its own .h file (and .cc file if you so desire) as it will be required after iteration 2.
- Updated March 3rd: Updated parser_tests.h
- Updated March 1st: Corrections to incorrectly specified locations for files

Specifications for integrating the parser.

For the second iteration of the project, you will incorporate a recursive descent top-down parser into your project. A number of classes are provided to you and you are to use them -- unchanged -- with your scanner implementation from iteration 1.

Five additional sample program files written in the Forest Cover Analysis Language(FCAL)), a test file (parser_tests.h), and other C++ (.cc and .h) files are provided. You can find all of these files in the the **class_repo** Git repository in a directory named `ProjectFiles/Iter_2_Files`. Copy the programs in the `samples` directory into the `samples` directory in your `project` directory in your group repository. The C++ files int the `src` directory should be copied into the `src` directory in your `project` directory, the header (.h) files in the `include` directory should be copied into the `include` directory in your `project` directory, and the files in the `tests` directory should be copied into the `tests` directory in your `project` directory.

I will go over the files in class on Tuesday 2/28, and your TA will discuss these files in lab on Friday 3/3.

One of your tasks is to extend your existing `Makefile` so that the new code is compiled and the new tests are successfully executed by the `run-tests` target produced by your updated `Makefile` .

Another task is to write a sample program in the Forest Cover Analysis Language (FCAL) language. The program will declare a boolean variable and use it in a while loop to sum the squares 1 through n together. So, when max = 5, the sum_of_squares is 1 + 4 + 9 + 16 + 25 (which equals 55).

The pseudo code for the sample program you should write is as follows:

```
sum_of_squares = 0
num = 1
max = 5
flag = true

while (flag) {
      sum_of_squares = sum_of_squares + num * num
      num = num + 1

  if (num > max) {
            flag = false
      }
}
```

Note, you must use the proper syntax from the FCAL language to create a syntactically correct program that implements the algorithm specified in the psuedo-code above. You do NOT have to adhere to the google style guide when writing FCAL programs.

You should save the program in a file named **mysample.dsl** , and make sure to put it in your `samples` directory. You are

responsible for writing the program with correct FCAL syntax so it parses correctly and passes the tests in `parser_tests.h` .

Also, you will have to make changes to your iteration 1 solution in order to get the new sample program you created to parse correctly. In response to the instructor's error in leaving out the boolean type and user-demanded requirements changes, we have updated the enumerated type `tokenEnumType` to contain the following entries: `kBoolKwd, kTrueKwd, kFalseKwd, and kWhileKwd` . The updated enumerated type is located in the file `iter2scanner.h` A version of the updated code is also below.

```
/*
 * This enumerated type is used to keep track of what kind of
 * construct was matched.
 */

enum kTokenEnumType {
  kIntKwd,
  kFloatKwd,
  kBoolKwd,
  kTrueKwd,
  kFalseKwd,
  kStringKwd,
  kMatrixKwd,
  kLetKwd,
  kInKwd,
  kEndKwd,
  kIfKwd,
  kThenKwd,
  kElseKwd,
  kRepeatKwd,
  kWhileKwd,
  kPrintKwd,
  kToKwd,

  // Constants
  kIntConst,
  kFloatConst,
  kStringConst,

  // Names
  kVariableName,

  // Punctuation
  kLeftParen,
  kRightParen,
  kLeftCurly,
  kRightCurly,
  kLeftSquare,
  kRightSquare,
  kSemiColon,
  kColon,

  // Operators
  kAssign,
  kPlusSign,
  kStar,
  kDash,
  kForwardSlash,
  kLessThan,
  kLessThanEqual,
  kGreaterThan,
  kGreaterThanEqual,
  kEqualsEquals,
  kNotEquals,
  kAndOp,
  kOrOp,
  kNotOp,

  // Special terminal types
  kEndOfFile,
  kLexicalError
};

typedef enum kTokenEnumType TokenType;
```

You must use the updated kTokenEnumType we have provided for you (above) when you update your scanner. In particular, you will have to update the files `scanner.h, scanner.cc, and scanner_tests.h` and possibly others you created in iteration 1 in response to the following changes:

1. The `kTokenEnumType` in your `scanner.h` file has been changed to facilitate the afrementioned requirements changes.
2. Update your scanner to recognize the keywords `boolean, True, False, and while` , and to create tokens for each of

the the keywords `boolean, True, False, and while` with the tokenTypes `kBoolKwd, kTrueKwd, kFalseKwd, and kWhileKwd` and add them to the list of tokens created by the scanner when they are in the input file being scanned.
3. Add 4 tests to the file `scanner_tests.h` that check to make sure that the keywords `boolean, True, False, and while` are recognized by the scanner, and that the scanner returns a token with the proper lexeme and enumerated type value when those keywords are recognized. **The tests should be similar to the other tests you wrote to test the other tokens in the FCAL language for iteration 1** .

This should go smoothly, but there may be some bumps along the way depending on your implementation of Iteration 1.

Division of labor statement

You are also required to document how you and your partner(s) split up the work of this iteration. You must specify what parts of the development and implementations each of you did, or if this work was done jointly. This document must reside in a plain text file named `iteration2_work.txt` that must be placed inside the `src` directory.

Due dates and submission and assessment procedures.

Your source code and test cases must be committed to your *team* Git repository and must be stored in the appropriate directory in your `project` directory. You must tag the code you want us to assess with the tag "iter2". Refer to the process for doing this for iteration 1 (lab 6) if you need to.

We will check out your work and run `make run-tests`, among other commands, from your `project` directory to assess your code.

The due date for this iteration is Friday, March 10 at 11:55pm. Your code and test cases must be committed; tagged properly, and pushed to `github.umn.edu` by this time.

This due date gives you approximately 10 days. Be advised that there will likely be other assignments and work to do in this time, so start early.

Assessment.

The following rubric will be used to compute your score for Iteration 2.

```
Possible points: 100.

Code Development Process: 60 points.
-------------------------------------

(These points are assessed once, on the due date.)

___ /5: tag "iter2" used properly.

Division of work statement (5 points).
___ /5: 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.

Creating the FCAL program and making the updates to the scanner (from iteration 1)
as described above (5 points):

___/2: Create the FCAL program as specified above in a file named **mysample.dsl** , and ensure it is in the samples directory.
___/3: Updates to the files scanner.h, scanner.cc, and scanner_tests.has specified above (and others as necessary).

Successful compilation and generation of the iteration 1 test cases
with the changes as specified above in place.
Each of the following commands succeeds with no errors (5 points).

___/1: make scanner.o
___/2: make scanner_tests.cc
___/2: make scanner_tests

Running updated test cases for iteration 1 (the scanner) doesn't result in a segmentation fault. (2 points)
Test cases may fail, but should not cause the testing framework to
crash.
___/2

Valid tests for new tokens are added to scanner_tests.h file pass when tests are run (4 points)
___/4

Successful compilation and generation of tests cases for the parser. (9 points)
Each of the following commands succeeds (i.e., executes) with no errors:
___/3: make parser.o
___/3: make parser_tests.cc
___/3: make parser_tests

Running 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: 40 points.

(These points will be assessed now and in the next iteration.
Thus, if you fail to pass some test cases now you can get those points back in iteration 3.)

___/5: When executed, parser_tests passes the test: test_parse_bad_syntax
___/5: When executed, parser_tests passes the test: test_parse_sample_1
___/5: When executed, parser_tests passes the test: test_parse_sample_2
___/5: When executed, parser_tests passes the test: test_parse_sample_3
___/5: When executed, parser_tests passes the test: test_parse_sample_4
___/5: When executed, parser_tests passes the test: test_parse_sample_5
___/5: When executed, parser_tests passes the test: test_parse_mysample
___/5: When executed, parser_tests passes the test: test_parse_forestLossV2