Iteration 1 - Building the first component of the Translator: The Scanner

Updates:

- Make sure to check this file requlary (i.e., at least daily) for updates!

Specifications for the scanner

For the first iteration of the project, you will build a scanner that takes as input a sample program in our domain specific programming language FCAL (Forest Conver Analysis Language), recognizes the tokens in the program, and returns them in a linked list.

You will write several test cases to demonstrate that your code works properly. In fact, you will not write a `main` function to create an executable program for this iteration. All execution of your scanner code will be done via test cases.

Several sample program files, a skeleton of a testing file, and other C++ files are provided. You can find all of these files in the public course Git repository (class_repo) in a directory named `ProjectFiles/Iter_1_Files`.

You should keep in mind that, like all software development projects, the requirements for the scanner may change some in later iterations. Thus, you should design and implement your code in a manner that such future changes will be not be too difficult to implement.

Required directory structure for the course project.

For lab 4 you created a directory `project` that contained a `src` directory. You should place your code in the directories specified below (which you should create if they don't exist in your teams `project` directory.

You will be given C++ files to put into the `src` directory in the `project` directory.

You will be given .h files to put into the `include` directory in the `project` directory.

You will be given .h files (with the word *test* in their name to put into the `tests` directory in the `project` directory.

Also there are some files in a `samples` directory. The `samples` directory, and the files in it must be placed inside the `project`directory (and thus next to the `src` directory).

A copy of the `cxxtest` directory must also be placed in your `project` directory, in a directory named `lib`. You should use the version of cxxtest that was given to you in lab 3. The `Makefile` provided indicates precisely where this directory and its contents must be located in your repository.

Structural requirements for the code.

There are some requirements on the data types and functions used in your implementation of the scanner. These will be utilized by the parser, which we will incorporate in the next iteration.

An enumerated type for kinds of tokens.

`tokenEnumType` is defined in scanner.h. Its definition is:

```
    enum kTokenEnumType {
/*0*/ kIntKwd,
```

```
    kFloatKwd,
    kStringKwd,
    kMatrixKwd,
    // and the rest of the legal terminal (or token) types for the FCAL language
    ...
    // Special terminal types (the last two terminal or token types
    /*39*/ kEndOfFile,
    kLexicalError
  };
```

Do **NOT** modify this definition. If you do, you will likely loose many points.

You must define the type name `TokenType` to refer to this enumerated type. This is done as follows:

```
typedef enum kTokenEnumType TokenType;
```

These are provided for you in the partially complete `scanner.h` file.

The values in this enumerated type (e.g., `kIntConst`, `kFloatConst`, *etc.*) must be kept as is so that they can be used by the parser in iteration 2 and by the test cases we will run in assessing your work.

A class for tokens.

You must define a class for tokens named `Token`. It must have the following fields:

- a field named `terminal_` of type `TokenType`,
- a field named `lexeme_` that is `string`.
- a field named `next_` whose type is a pointer to the `Token` class type `Token`. The `next_` attribute will be used for creating a singly-linked list of tokens that will be passed to the parser. Lists formulated using the `scan` method of the `Scanner` class you will design and implement are used in some of the tests in the `scanner_tests.h` file provided to you.

All of these fields must be **private**, and you will define the accessor (get) and mutator (set) functions for manipulating the values of a Token.

The `Token` class *can* have additional fields however, for example, keeping keeping track of the length of the lexeme, or the line and column number on in the file where the lexeme was found.

A class for the scanner.

You must also define a class named `Scanner` that has a `scan` method of the following type:

```
Token *Scan (const char *) ;
```

This method is used in some of the test cases.

Behavioral requirements for the code.

While most of the behavioral requirements will be clear from the test cases provided, a few additional comments included below may be helpful.

1. All lists returned by the scanner should end with a token object whose terminal is `kEndOfFile`
2. All lists should contain one and only one token object whose terminal is `kEndOfFile`
3. When a lexical error occurs, the scanner should create a token with a single-character lexeme and `kLexicalError` as the terminal.

Tasks.

There are three main tasks for you to complete in this iteration. The first is to design your scanner - for example, by writing pseudo-code to specify the function of the Scan method - we will not evaluate your design at this time. The second it to write an adequate set of tests that exercise your code and the third is to write the code that will pass the tests you create (the minimum set of tests required are described in the assessment section below), and the tests we have provided.
Morever, groups need to separate test and implementation - that is, if one person implements a feature, the other writes the tests for it, rather than one person doing both. Your git commit messages should reflect this.

Writing tests

In the file `scanner_tests.h` provided you will find some comments instructing you to write test cases that exercise the functions and methods that are called by your `scan` method. The design and implementation for these is up to you. The only requirement is that these supporting functions and methods provide significant support to the `Scan`. Your test cases for for the supporting functions and methods should properly test the code using structured basis testing.

The goal of writing and testing supporting functions and methods is to significantly simplify the writing and testing of the `Scan` method. We discussed testing length in a number of lectures (and labs) so you should refer to your notes from those lectures and labs. We will revisit the topic as well.

Writing the code

The specifications in this document primarily address this task.

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 design, implementation, and test each of you did, or if this work was done jointly. You are strongly encouraged to work together simultaneously (in the same location or remotely connected) when doing all three activites. You must also specify what parts of the code and tests each of you have completed - individually or jointly (and if jointly, perhaps using pair programming techniques) in both this document and by documenting your commits in git. The division of labor document must reside in a plain text file named `iteration1_work.txt` that must be placed inside the `project` directory.

In any case, your commits to you project git repository should correspond to the code you have written, and you will be graded this as well.

Due dates and submission and assessment procedures.

You source code and test cases must be committed to your *team* Git repository and must be stored in the appropriate directory (`project`,`project/src/`, `project/include`, `project/tests`), or `project/samples` . An additional requirement of creating a new branch for the work that you turn in will be discussed in lecture or lab and the specifics will be accessible on moodle. Stay tuned.

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

- More specifically, when we evaluate your code we will run the following command:

  ```
  make run-tests
  ```

  Make sure that your Makefile has this as its target for running the tests.

- Moreover, We will run the test cases that you have in your modified `regex_tests.h` and `scanner_tests.h` files, but we will ALSO run the tests in the original `regex_tests.h`, `scanner_tests.h` file that we gave you (and possibly other tests we have not given you). Thus, you **MUST NOT** modify the enumerated type given to you in `scanner.h`.

The due date for this iteration is Tuesday, February 28 at 11:55pm. Your code, test cases, and division of work statement must be committed by this time.

Assessment.

The following rubric will be used to compute your score for Iteration 1.

```
Possible points: 150.

Code Development Process: 90 points.
-----------------------------------
(These points are assessed once, on the due date.)

Correct use of Git (12 points).

Make sure each of your directories and
files is in the correct place in your team git repository. Make proper
use of .gitignore - that is, no executable files, .o files or files with
a trailing ~ are committed. Finally, make sure to commit your changes
frequently, and use detailed messages to describe what you have done.

___ / Correct use of Git (12 points).

Division of work statement (8 points).

___ / 8: A well written, grammatically correct, spell-checked
         and non-trivial division of work statement.

Successful compilation and generation of tests cases.  (9 points)
Each of the following command succeeds with no errors.
___ / 2: make regex_tests
___ / 2: make scanner.o
___ / 2: make scanner_tests.cc
___ / 3: make scanner_tests

Code adheres to the google style guidelines (10 points)

___ / 5: code passes the google style checker
___ / 5: Code adheres to the following google style guidelines(unless specified 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

There is a test case for each terminal symbol type in the enumerated
type TokenType.  (41 points)

___ / 41: a test case for each item in TokenType, 1 point each
          For a terminal named X, a test case named test_terminal_X
          must be present.  For example, for the kFloatConst terminal
          symbol (a value in the TokenType enumerated type) you must
          have a test named test_terminal_floatConst.


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: 60 points.

(These points will be assessed in the next iteration as well and thus if you fail to
pass some test cases now you can get those points back on iteration 2.)

___ / 41: passes a test case for each item in TokenType, 1 point each
For a terminal named X, a test case named test_terminal_X must pass present. For example, for the floatConst terminal symbol (a value in the TokenType enumerated type) you must have a test named test_terminal_floatConst that passes.

___ / 2: passes the test test_scan_empty

___ / 2: passes the test test_scan_empty_comment

___ / 2: passes the test test_scan_lexicalErrors

___ / 3: passes the test test_scan_nums_vars

___ / 5: passes the test test_scan_bad_syntax_good_tokens

___ / 5: passes the test test_scan_sample_forest_loss_v2