CSCI 4160 Project2

Due: see class calendar

Goal:

This assignment serves several purposes:

- to be familiar with flex
- to understand lexical analysis using the existing tool

Description:

In this assignment, you are required to use flex – a scanner generator to generate a scanner for Tiger language. The manual for Tiger language can be found at the class repository: Tiger\manual.pdf.

Tiger language:

The reserved words of the language are:

while, for, to, break, let, in, end, function, var, type, array, if, then, else, do, of, nil.

The punctuation symbols used in the language are:

*, : ; ( ) [ ] { } . + - * / = <> < <= > >= & | :=*

The string value that you return for a string literal should have all the escape sequences translated into their meanings.

There are no negative integer literals; return two separate tokens for -32.

Detect unclosed comments (at end of file) and unclosed strings.

Tips and Requirements:.

Your lexer should use regular expressions to do the work EXCEPT for the nesting counting. For example, don't find a beginning quote and then use C++ code to look for matching end quote… use regular expression(s)….

The comments in Tiger language is the same as C style comments that start with /* and end with the matching subsequent */. Any symbol is allowed within a comment. However, the comments can be nested, i.e.,

```
/* this is a line
   /* this is ok
As many lines as you desire
This one only has three
  */  this closes the nested comment
*/  // this closed the first one.
```

Nesting can be of any level. Comments are ignored similar to whitespace. To handle nested comments, you will need a counter within lexer that counts the beginning of comment and decrements when ending. The comment is completed when it reaches zero.

String literal is a sequence, between quotes (“), of zero or more printable characters, spaces, or escape sequences. Each escape sequence is introduced by the escape character \, and stands for a character sequence. The allowed escape sequences are as follows (all other uses of \ being illegal):

- \n a character interpreted by the system as end-of-line.

- \t TAB
- \" the double-quote character (")
- \\ the backslash character (\)

What to do in this project?

You need to provide rules to recognize **reserved words** (*while, for, to, break, let, in, end, function, var, type, array, if, then, else, do, of, nil)*, **punctuation symbols** (*, : ; ( ) [ ] { } . + - * / = <> < <= > >= & | :=*), **comments**, **identifiers**, **integer literals**, and **string literals**. More specifically,

- For white space character like " ", \n, and \t; recognize them and discard it.
- For each recognized reserved words , and punctuation symbols, return their corresponding tokens in the rule action. All tokens are defined in tokens.h file.
- For comments, just ignore all content in the comments. Make sure your rules recognize nested comments.
- For integer literal, recognize it and return the INT token. Make sure you store the associated integer value to variable *yylval.ival*.
- For identifier, recognize it and return the ID token. Make sure you store the associated name to variable *yylval.sval*.
- For string literal, it is better to use start condition
    - For the double quote (") marking the beginning of the string literal, reset the variable *value* to be an empty string, modify variables beginLine and beginCol, and start STRING condition;
    - For all rules under STRING condition that recognize part of the string literal, append the recognized part to the variable *value*;
    - For all rules under STRING condition that finds an error (like illegal escape sequence, unclosed string), report the error.
    - For the closing double quote ("), copy variable *value* to *yylval.sval*, just like the actions in identifier rule, and start the INITIAL condition.

Make sure the provided function **newline()** is called for every newline character in the source file. Make sure the provided function **error()** is called for every recognized errors (illegal character, unclosed comments/strings, illegal escape character sequences, etc.). **Other than calling function error(); there should be no output statement in rule actions.**

Instructor provided files in the class repository

The following files are provided by the instructor:

- Folder FlexProject. This contains a sample Visual Studio 2010 project. If you want to use this provided project for this assignment instead of creating your own project from scratch, please pay attention to the following:
    - You have installed the Flex and Bison as specified in the class repository document:
      How-to\flex and bison\ flex and bison installation.doc
      Make sure the folder (like c:\gnuWin32\bin) containing flex.exe/bison.exe is added to your path.
    - Make sure instructor provided FlexLexer.h and flex.skl are copied to the correct location as specified in the first 2 pages of the following document in the class repository:
      How-to\flex and bison\windows\ HOWTO-C++-flex-bison-Visual Studio.doc
    - Skeleton source files provided in the sample project are listed below:

    - tokens.h: contains the definition of all tokens and definition of YYSTYPE, which is a structure to hold values associated with matched token.
    - ErrorMsg.h: contains the definition of error handler
    - main.cpp: the driver
    - tiger.ll: a flex skeleton file for this project. In this assignment, you only need to work on this file. No change on other files is necessary.
    - Test0.tig, test1.tig, test2.tig: different test program of tiger language
- Description2.pdf: this file
- Rubric2.doc: the rubric used to grade this assignment.
- Test0.txt, test1.txt, and text2.txt. These are instructor provided output for test0.tig, test1.tig, and test2.tig, respectively. Your output may be different depending on how you handle string errors.

HOW TO GET STARTED

1. Pull from class repository on ranger to your local class repository. All files provided by the instructor can be found at projects/project2.
2. Create master repository on ranger as specified on "How to prepare for projects"….see class repository.
3. Clone your project2 master repository on your local machine, and copy folder FlexProject to your local repository of project2.
4. Add folder FLexProject to git, and commit, and push it to master repository.
5. After making significant progress, make sure you submit the change to local repository and push all the changes to master repository on ranger.
6. Finish coding and debugging...remember to add comments and commit as needed.
7. Once you have finished, submit the project in the following way:
    - Copy the file projects/project2/rubric2.doc from the class repository to the project2 folder in your local repository of project2. Edit the file to put your name.
    - Commit the whole project2 folder to your local repository.
    - Push all the changes to master repository on ranger.
    - **Any commit of the project after the deadline is considered as cheating. If this happens, the latest version before the deadline will be graded, and you may receive up to 50 points deduction.**
8. You can check your grade by update the rubric2.doc from the repository after the notice from the instructor.