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
- to create a scanner for COOL using flex.

Description:

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

COOL language:

Please read COOL manual carefully to understand the language. Some highlights of COOL lexical structure is given below:

- The lexical units of Cool are integers, type identifiers, object identifiers, special notation, strings, keywords, and white space.
- Integers are non-empty strings of digits 0-9.
- Identifiers are strings (other than keywords) consisting of letters, digits, and the underscore character. Type identifiers (i.e. class names) begin with a capital letter; object identifiers (i.e. object, attribute, and method names) begin with a lower case letter. There are two other identifiers, self and SELF TYPE that are treated specially by COOL but are not treated as keywords.
- There are two forms of comments in COOL. Any characters between two dashes "--" and the next newline (or EOF, if there is no next newline) are treated as comments. Comments may also be written by enclosing text in (* . . . *). The latter form of comment may be nested.
- The keywords of cool are: **class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true**. Except for the constants true and false, keywords are case insensitive. To conform to the rules for other objects, the first letter of true and false must be lowercase; the trailing letters may be upper or lower case.

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 multiple-line comments in COOL 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 (\)

We maintain three global tables as defined in StringTab.h file to eliminate duplicate copies of literals and identifiers. These three tables are:

- identifier table (global variable name: idTable): store all type and object identifiers;
- string table (global variable name: stringTable): store all string literals;
- integer table (global variable name: intTable): store all integer literals.

Whenever an identifier or a string/integer literal is detected, you need to insert it into the corresponding literal table and assign the return value to *yylval.symbol* before returning the token.

What to do in this project?

You need to provide rules to recognize **key words**, **punctuation symbols**, **operators**, **comments**, **type and object identifiers**, **integer literals**, and **string literals**. More specifically,

- For white space character like “ ”, \n, and \t; recognize them and discard it.
- For each recognized key word, punctuation symbol, and operator, return its corresponding token in the rule action. All multi-character tokens are defined in tokens.h file. If the punctuation symbol/operator is represented by a single character, just return that character.
- For comments, just ignore all content in the comments. Make sure your rules recognize nested comments.
- For integer literal, recognize it and return the INTCONST token. Don’t forget to update *yylval.symbol* before returning the token.
- For identifiers like type name and variable/method name, recognize it and return the TYPEID or OBJECTID token. Don’t forget to update *yylval.symbol* before returning the token.
- For Boolean constants, update *yylval.boolean* to true or false before returning BOOLCONST token.
- For string literal, it is better to use start condition
    - For the double quote (“) marking the beginning of the string literal, reset the variable *buffer* 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 *buffer*;
    - For all rules under STRING condition that finds an error (like illegal escape sequence, unclosed string), report the error, and recover from the error if necessary.
    - For the closing double quote (“), insert the variable *buffer* into string literal table and assign the return value to *yylval.symbol*. Then start the INITIAL condition and return STRCONST token.

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.**

Where to download files for this project?
The project is provided for two different platforms: Windows and Linux (through Jupyterhub). To run it on Linux, you need to enter the folder containing the project and type the following command:
--work on COOL.ll file
make --compile your project using make command
./main cool0.cl --run your program against cool0.cl, which can be replaced by other COOL files

If you want to use MacOS for the project, you can use the Linux version. Please make sure latest version of Flex and Bison are installed on your machine. You may need to change "g++" at line 9 of **makefile** to the compiler you want to use. The compilation and execution should be similar to the Linux version.

If you want to know more about make and makefile, please check this page.

Instructor provided files

The following files are provided by the instructor:

- 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
  - StringTab.h and StringTab.cc: contains definition of different literal tables.
  - main.cpp: the driver
  - cool.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.
  - cool0.cl, cool1.cl, cool2.cl: different test programs of COOL language
- Description2.pdf: this file
- Rubric2.doc: the rubric used to grade this assignment.
- cool0.txt, cool1.txt, and cool2.txt. These are instructor provided output for cool0.cl, cool1.cl, cool2.cl, respectively. Your output may be different depending on how you handle string errors.

What/How to submit
Please submit COOL.ll file only to D2L dropbox of this project.