Lab 2: Lexical Analysis

Description

Use flexc++ to implement a lexical scanner for the Tiger language. Appendix A describes, among other things, the lexical tokens of Tiger.

Flexc++ is a tool for generating lexical scanners: programs recognizing patterns in text. It reads on or more *inputs* files, containing rules: regular expressions, optionally associated with **C++** code. From this **Flexc++** generates several files, containing the declaration and implementation of a class (Scanner by default). The member function lex is used to analyze input: it looks for text matching the regular expressions. Whenever it finds a match, it executes the associated **C++** code.

Your lexical scanner for the Tiger language should include the following parts:

- Basic specifications;
- Comments handling;
- Strings handling;
- Error handling;

You will code in the directory *src/tiger/lex* and modify two files *tiger.lex* and *scanner.h*. We have already given some hints for you in the comments. Other related files for this lab are *src/tiger/errormsg/errormsg.** and *src/tiger/main/test_lex.cc*.

This chapter has left out some of the specifics of how the lexical scanner should be initialized and how it should communicate with the rest of the compiler. You can learn this from the flexc++ manual.

Notice: Before you start this lab, you should carefully read the chapter 2 of the textbook, and you may need to get some help from the flexc++ manual and the *Tiger Language Reference Manual* (Appendix A). As usual, if you have any question about this lab, feel free to contact teaching assistants.

Important Notes

1. You will need to code in *tiger.lex* and maybe in *scanner.h*. The file *scanner.h* is just the class declaration of your scanner, which is initially generated by flexc++ when they are not existing at the first time. Flexc++ will also generate *lex.cc and scannerbase.h*, which implement your scanner class according to your rules defined in *tiger.lex*.

2. The generated scanner class has many member functions that you can directly call in the rules of *tiger.lex*. We have also provided some helper functions (*adjust* and *adjustStr*) in *scanner.h*. You can define your own variables and functions in it.

3. Note that you don't need to set the sementic value of each token like the codes(*yylval*) in the textbook because we have already set it for you in *parser.ih*. What you need to do is return each token's enumeration value.

4. For string tokens, you are required to convert the string literal to the real string value and set your scanner's current matched to it using *setMatched()* member function. For example, a matched string literal maybe *"hello\n"*, your should convert the escape *"\n"* to the real linebreak character.

Hints

• Scanner member functions that you may use:
  1. *matched()*
  2. *setMatched()*
  3. *begin(StartCondition__::COMMENT)* **Notice**: The enum type *StartCondition__* has **two** underlines.

• Refer to Start Conditions for comment handling and string handling.

Documentation

Along with your code, you should turn in documentation for the following point:

• how you handle comments;

• how you handle strings;

• error handling;

• end-of-file handling;

• other interesting features of your lexer.

You are supposed to create a directory named doc under your repo. Put your documentation(in pdf or md) under that dir and push it along with your code. We sugguest you keep your documentation neat.

Environment

You will use the same code framework that you had set up when you worked on lab1. What you need to do now is to pull the latest update of the code framework if there are any. You may have to do some code merging jobs.

```
shell% git fetch upstream
shell% git checkout -b lab2 upstream/lab2
```

```
shell% git push -u origin
shell% git merge lab1 (some students may finish lab1 on master, thus merge master instead)

You may have to do some code merging jobs here, we strongly suggest you use commands below to merge your file

shell% git checkout --ours file
After merging, you are supposed to push your update to your remote repo
shell% git add files
shell% git commit -m "[lab2] merge lab1"
shell% git push origin lab2
```

If you haven't set it up before, you should follow the instructions here to set up your lab environment.

Grade Test

The lab environment contains a grading script named as **grade.sh**, you can use it to evaluate your code, and that's how we grade your code, too. If you pass all the tests, the script will print a successful hint, otherwise, it will output some error messages. You can execute the script with the following commands.

```
Remember grading your lab under docker or unix shell! Never run these commands under windows cmd.
shell% make gradelab2
shell% ...
shell% [^_^]: Pass #If you pass all the tests, you will see these messages.
shell% TOTAL SCORE: 100
```

Handin

The deadline of this lab is on Friday 12:00 AT NOON, Sep. 29, 2023, and if you miss the deadline, points will be deducted based on the number of days you are late!

After you have passed the grade test, you need first commit your modification, then push it to your remote repository on gitlab. You can use the following commands to finish this step.

```
shell% git add src/tiger/lex/tiger.lex src/tiger/lex/scanner.h
shell% git commit -m "[lab2] finish lab2"
# If you have registered in lab1, no need to register again as long as .info file is still in your root dir, otherwise make register before push.
shell% git push origin lab2
shell% # Verficate your ID and password#
```
