CSE 340 Spring 2016 Project 2

Due: Feb. 1, 2016 by 11:59:59 pm

1. Introduction

You will be given a lexer that reads tokens from standard input. Your goal is to write, in C or C++, a program that reads all tokens from standard input by calling the lexer function `getToken()` and storing certain tokens in a linked list. After all tokens are read, your program will print out the content of the linked list in a specific order.

The next section describes the lexer API. You also need to read the provided code in `lexer.c` and understand how the lexer works. You will only use the `getToken()` function in this project.

2. Lexer API

There are two functions that the lexer defines. These two functions compose the application programming interface (API) of our lexer. These functions are declared in `lexer.h` (and implemented in `lexer.c`). You will find the files `lexer.h` and `lexer.c` on the submission site for project 2.

- `getToken()` reads the next token from standard input and returns its type as a `token_type` enum. If the token is of type `ID`, `NUM`, `IF`, `WHILE`, `DO`, `THEN`, or `PRINT`, then the actual token value is stored in the global variable `current_token` as a null-terminated character array and the length of the string is stored in the global variable `token_length`. There are two special `token_type` values: `END_OF_FILE`, which is returned when the lexer encounters the end of standard input and `ERROR`, which is returned when the lexer encounters an unrecognized character in the input.

- `ungetToken()` causes the next call to `getToken()` to return the last token read by the previous call to `getToken()`. Note that this means the next call to `getToken()` will not read from standard input. It's a logical error to call `ungetToken()` before calling `getToken()`. This function is useful for writing recursive descent parsers that you will see later on in this course.

There are four global variables declared in `lexer.h` that are set when `getToken()` is called:

- `t_type`: the token type is stored here. Note that this will be the same value that was returned by `getToken()`.

- `current_token` : the token value is stored in the array `current_token` . If the token is of type `ID` , `NUM` , `IF` , `WHILE` , `DO` , `THEN` , or `PRINT` , then `current_token` contains the token string. For all other token types, `current_token` contains the empty string.
- `token_length` : the length of the string stored in `current_token` .
- `line` : the current line number of the input when the token was read.

You should read the source code provided in `lexer.c` and `lexer.h` . Here is a hint for using the lexer: you can use the token type labels such as `NUM` or `END_OF_FILE` directly in the code. For example, if you want to check if the token type is `NUM` , you can write the following code:

```
if (t_type == NUM)
{
    // ...
}
```

3. Requirements

Your program should use the provided lexer and read all tokens from the input by repeatedly calling the `getToken()` function. Certain token strings and additional data should be stored in a linked list. Specifically, if either of the following conditions are true:

- The token is of type `NUM`

  OR

- The token is of type `ID` AND the actual token is equal to one of the following values: `"cse340"` , `"programming"` , or `"language"`

Then the token string and other information needs to be stored in a node of a linked list. The information that needs to be stored about each of these tokens in the linked list is the following:

- Token type (from `t_type`)
- Token value (from `current_token`)
- Line number of the input where token was read (from `line`)

After reading all tokens from the input and storing information about tokens that match the criteria, your program should go over the linked list and print the information in **reverse order** from when that token was encountered. Each of the tokens in the linked list must be printed to standard output on a separate line with the following format:

```
<line> <token_type_string> <token_value>
```

Note that `<token_type_string>` is the textual representation of the token type. In this case, the possible values are `ID` and `NUM`.

You should write all your code in a separate C or C++ file and include `lexer.h` to be able to access the lexer functions. Here is how to do it in C:

```c
#include "lexer.h"

int main()
{
    // TODO: write your code here!
    return 0;
}
```

And this is how to do it in C++:

```cpp
extern "C"
{
  #include "lexer.h"
}

int main()
{
    // TODO: write your code here!
    return 0;
}
```

To compile your code, you should use the GCC compiler from CentOS 6.7 that you have installed from last project. This is how to compile your code if you are writing it in C:

```
gcc lexer.c your_code.c
```

Where `your_code.c` should be replaced by the file name where you store your code.

To compile your C++ program, use the following two commands instead:

```
gcc -c lexer.c
g++ lexer.o your_code.cpp
```

NOTE: You should not modify `lexer.c` or `lexer.h`. Write your code in a separate file as described above.

Example

Here is an example input with four lines:

```
cse340 < < + 123 *
456 programming
- cse 340 , LANGUAGE 100
. ; WHILE 200 IF
```

Here is the expected output:

```
4 NUM 200
3 NUM 100
3 NUM 340
2 ID programming
2 NUM 456
1 NUM 123
1 ID cse340
```

Notice that the tokens are listed from last to first (reverse order).

4. Testing

This section is relevant to this project as well as *all later projects* in this course.

4.1. Input / Output

Your program should read the input from **standard input**. That is normally the keyboard input and it can be accessed by using many standard C functions like `getchar()`, `scanf()` etc. In C++, you would use `cin` to read from standard input. In this project, you won't read the input directly, the call to `getToken()` would read the input by using `getchar()`. Read the source code in `lexer.c` to see how this is done.

Your program should write its output to **standard output**. That is normally done by using such functions as `printf()` or `puts()` in C or using `cout` in C++.

4.2. Testing Your Code

Our grading system is automated hence another program (running on the submission site server) tests your code for correctness. We provide exact input/output format for every project in addition to multiple `test cases`. A test case is composed of two parts:

- A specific input in a file with `.txt` extension
- The expected output in a file with `.txt.expected` extension

Your program **passes** a test case if given the input in the `.txt` file, it produces exactly the same output as the contents of the respective `.txt.expected` file. Otherwise your program **fails** that test case.

4.2.1. Testing with a single test case

To feed the input file to your program without typing its content on the keyboard, we use **input redirection** and you should too! Here is how you would redirect the standard input of a program to a file:

Let's assume that we have a compiled program named `a.out` (an executable file). Normally, you would run this program like this (in a terminal):

```
./a.out
```

Which would wait for you to provide the input by typing it on the keyboard (that is if the program expects any input). To redirect the standard input to a file named `test01.txt`, you should run the program like this instead:

```
./a.out < test01.txt
```

The "less than" character instructs the command interpreter to read the file specified after the `<` and feed it to the program through its standard input.

Similarly, we can redirect the standard output to a file. This way instead of printing the output of the program on the terminal, the output will be stored in a file. To redirect the standard output, we use the "greater than" character with a file name like this:

```
./a.out > output.txt
```

Which will store the program's output in `output.txt`.

NOTE: if a file with the same name exists, it will be overwritten!

Of course you can mix input and output redirection as well:

```
./a.out  < test01.txt  > output.txt
```

Our automated grading system compares the output generated by your program with the expected output by using the `diff` command. `diff` is a system utility that is used to compare the contents of two files. If the files are different, it produces a report highlighting the differences otherwise it outputs nothing. We use `diff` with `-Bw` option which causes it to ignore differences in whitespaces. Here is an example of how to compare the output generated by the program with the expected file:

```
diff -Bw  output.txt  test01.txt.expected
```

4.2.2. Running multiple test cases

You can also use the shell script that we have provided called `test1.sh`, which automates this process for multiple test cases. The test script assumes that your compiled program is called `a.out` and test cases are stored in a subdirectory named `tests`. You can run it like this:

```
./test1.sh
```

If you see the error message `-bash: ./test1.sh: Permission denied`, then run the following command to fix the problem:

```
chmod +x test1.sh
```

The test script runs your program with I/O redirection for each test case in the `tests` folder and compares the output with the expected file using `diff` and reports any differences for failed cases. At the end it will print the number of tests passed and removes any temporary files created.

It is strongly recommended that you test your program before submission with the provided test cases the way I described here.

5. Evaluation

Your submission will be graded on the number of test cases passing when you submit your code. If your program does not compile on the server, you will not receive any credit, so if you a see a compiler error message on the submission site, fix the problem and submit again.

6. Submission

Submit your code on the course submission site. You should only upload **your source code**. You should not upload the compiled program or test cases. You should not upload `lexer.h` or `lexer.c`, these files will be automatically added to your submission. You should not use space in your file names.

Submit your project here before the deadline: https://cse340.fulton.asu.edu/cse340/