CSE340 Spring 2015
Project 1

Due by 11:59 PM Tuesday Jan. 27, 2015

Abstract

The goal of this project is to explore strings, pointers, and memory allocation in C and familiarize you with the platform and tools that will be used in all programming assignments in this course. You will use a lexical analyzer that we developed to learn about strings, linked lists and memory management in C. You need to master these topics in order to have a much easier time in subsequent projects.

1 Introduction

You are provided with a small lexer (lexical analyzer) that reads tokens from standard input. Your job is to write a program that reads all tokens from the input by calling the lexer function `getToken()` and store some tokens in a linked list and then print out the contents of the linked list in a specific order.

The following section describes the provided code in detail. You should read it carefully to learn how to use it in your code.

2 Lexer API

The lexer is composed of several functions most of which are only meant to be called by the lexer itself and only two of the functions are intended to be used in other code. In other words, these two functions provide the application programming interface (API) of our lexer. These functions are declared in `lexer.h`:

- `getToken()`: reads the next token from standard input and returns its type as an int. In some cases the actual token is stored in a global variable named `token` which is a null-terminated character array. The cases for which the actual token is stored in `token` are: `ID`, `NUM`, `IF`, `WHILE`, `DO`, `THEN` and `PRINT`. The possible values for token type returned by `getToken()` are defined as macros in `lexer.h`. The special value `ERROR` signifies that `getToken()` has encountered an unrecognized character in the input (a lexical error) and `EOF` signifies the end of input stream.

- ⋆ `ungetToken()`: causes the next call to `getToken()` to return the last token read by the previous call to `getToken()`. It simply sets a flag so that next call to `getToken()` would not read another token from the input but rather return the last token. Note that it is an error to call `ungetToken()` before any call to `getToken()`.

There are 4 global variables declared in `lexer.h`:[1]

- ⋆ `ttype`: when `getToken()` is called, the token type is stored as an integer in the variable `ttype`.
- ⋆ `token`: when `getToken()` is called, the token *value* is stored in the array `token`. If the token is one of `ID`, `NUM`, `IF`, `WHILE`, `DO`, `THEN` or `PRINT`, the variable `token` contains the token string. For other token types, the variable `token` contains the empty string.
- ⋆ `tokenLength`: the length of the string stored in `token`
- ⋆ `line`: the current line number of the input

3 Requirements

Your program should use the provided lexer and read all tokens from the input by repeatedly calling the `getToken()` function. Some of the token strings should be stored in a linked list. Specifically, if

- ⋆ 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"`

the token string and some other information need 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 (copied from `ttype`)
- ⋆ Token value (copied from `token`)
- ⋆ Line number in the input where token was read (copied from `line`)

After reading all tokens from the input and storing information about tokens that match the above criteria, your program should go over the linked list and print the information **in reverse order** with the following format for each node of the linked list printed in a separate line:

[1] These variables are declared as `extern` variables which enables code in other files that `#include lexer.h` to *see* these variables. The actual variables are declared in `lexer.c`.

```
<line> <ttype_str> <token>
```

Note that `<ttype_str>` is the textual representation of the token type. The possible values are `ID` and `NUM`.

Your submissions will be inspected by the TA to give you feedback on potential errors in your code. You are required to store the above information in a linked list (either single or double linked list). The nodes in the linked list should be allocated on the heap (using `malloc` or other similar functions like `calloc`) and the allocated memory should be freed after printing the output.

3.1 Example

Here is an example input with 4 lines:

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

And 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
```

4 Grading

An important part of this project is to make sure that you implement the requirements in the manner specified. We will make sure that the output is correct, but we will also check to make sure that you implemented a linked list, allocated space correctly, manipulated strings correctly and so on. If the implementation does not follow the requirements, you will not get full credit for your work.

The course website which should be used to submit your programs, has a number of test cases for this project as well as future projects. Make sure your code passes all test cases. The course website is only accessible from inside the ASU network,

so you should either be on campus or connect to ASU network via VPN software in order to access the website. **The URL for accessing the website will be published on Blackboard**.

5 Submission

1. Only C or C++ can be used for this project
2. You should submit your code on the course website by 11:59:59 pm on due date. The website link will be posted on Blackboard
3. Read the provided code carefully, you need to fully understand it since it is required for this project and it is used in future projects
4. There are compiling instructions in the comments written at the end of `lexer.c`
5. Make sure your submission has no compile problems. If you get a compile error on the website, fix the problem and submit again
6. You can submit any number of times you need, but remember that we only grade your last submission
7. Write your code in a separate file, include `lexer.h` in your code to be able to use lexer functions
8. Don't submit the provided code, only submit the code that you write. `lexer.h` and `lexer.c` will be added to your submission automatically on the server
9. Don't include test scripts or test cases with your submission
10. Don't use any whitespace characters in file names