CS 214: Systems Programming, Fall 2020
Assignment 0: C Tokenizer

0. Introduction

In this assignment, you will practice programming with C pointers. Much of the pointer manipulation will come in the form of operating on C strings. A program that is at the base of every terminal, shell and interpreter is a tokenizer. For this assignment, you will write a tokenizer.

A tokenizer is a program that takes a single string of characters of an arbitrary length and breaks them up into types of inputs. For instance, the single string “blah 8 stuff 1.3” contains two 'word'-type inputs, an 'integer'-type input and a 'float'-type input. Your tokenizer's job will be to to take that string and break it up into each type of token, printing out each as it identifies them.

You will not know the length of the input string beforehand, so you will not know how much memory to allocate in your code to copy the input string. Since you do not know the length of the input string, you will not know the index of the string's last character. These are two fundamental issues you must consider.

1. Implementation

Your main() should take a single string as input and inspect that string, flowing left to right and printing out each token and its type, once the token's end is found. You may not modify the input string. You especially can not remove characters from it. Please create as many other functions as you need in whatever configuration you wish. Please investigate the ctype.h and string.h libraries for useful functions, however you may not use strtok().

A token is defined as a sequence of characters terminated either by a delimiter or the beginning of a new token. A delimiter is a special character or class of characters that denote the end of a token. For this assignment you can consider all 'whitespace' characters to be delimiters. Whitespace characters include a space (' '), tab ('\t'), vertical tab ('\v'), form feed ('\f'), newline ('\n') and carriage return ('\r'). Any of these will terminate a token. For instance:

“123 stuff”

Consists of two tokens. If you are scanning characters from left to right, once you hit the space at index 3, you can presume that the current token has stopped and the next character belongs to the next token. In your case the start of a new token can also terminate a token. For instance:

“123stuff”

Consists of two tokens. If you are scanning characters from left to right, once you hit the letter at index 3, you can presume that the current *decimal integer*-type token has stopped and the next character belongs to the next token.

There are several types of tokens:

A *word* token consists of an alphabetic character followed by any number of alphanumeric characters. Some word tokens are “hi” “stuffF” “ThinGs” “a1231” “C0MP14IN”

A *decimal integer* is a digit character followed by any number of digit characters: “12” “8” “2”

An *octal integer* is a '0' character followed by any number of octal digits (0-7).

A *hexadecimal integer* is “0x” (or “0X”) followed by any number of hexadecimal digits (0-9, a-f, A-F).

A *floating point* token is a *decimal integer* that has a '.' at any position other than the last. It may optionally contain an exponent in scientific notation at the end, for example: “3.14159e-10”

A C operator is any of the operators listed on the C reference card attached to the assignment. Its token type is the name of the operator. For instance the type of the token '[' is “*left bracket”.* There are at least 45 different operators and punctuation marks in C. Some of the operators consist of multiple characters. Each operator and punctuation mark is a distinct token.

You should also implement a main() function that takes a single string argument, as defined above. The string contains zero or more tokens. Your main() function should take a single string as input and print out all the tokens in it in left-to-right order. Each token should be printed on a separate line. For example:

./tokenizer “array[xyz] += pi 3.14159e-10”
word: ”array”
left bracket: ”[”
word: ”xyz”
right bracket: ”]”
plusequals: ”+=”
word: ”pi”
float: ”3.14159e-10”

Keep in mind that coding style will affect your grade. Your code should be well-organized, well-commented, and designed in a modular fashion. In particular, you should design reusable functions and structures and minimize code duplication. You should always check for errors. For example, you should always check that your program was invoked with the correct number of arguments.

Your code should compile correctly (no warnings and errors) with no flags other -o. For example:

```
$ gcc -o tokenizer tokenizer.c
```

should compile your code to a debug-able executable named tokenizer without producing any warnings or error messages. (Note that -O and -o are different flags.)

Your code should also be efficient in both space and time. When there are tradeoffs to be made, you need to explain what you chose to do and why.

IMPORTANT NOTE: Your submission MUST (un)tar (see below), compile and execute on the iLab machines or a zero grade will be given. Be sure to test your code on an iLab machine before handing it in.

2. Submission Requirements

Turn in a tarred gzipped file named “Asst0.tgz” that contains a directory called Asst0 with the following files in it:

A tokenizer.c file containing all of your code.

A file called testcases.txt that contains a thorough set of test cases for your code, including inputs and expected outputs.

A readme.pdf file that contains a brief description of the program and any great features you want us to notice.

Creating the tar file:
First create a directory named “Asst0” and copy your three files into it.
Then run “tar czvf Asst0.tgz ./Asst0”
This should create a new file named “Asst0.tgz” that contains your directory and your code.

Your grade will be based on:

Correctness (how well your code works).
Quality of your design (did you use reasonable algorithms).
Quality of your code (how well written your code is, including modularity and comments).
Efficiency (of your implementation).
Testing thoroughness (quality of your test cases).

2.1 Testing Note

When testing, please do not use the characters:

\ (backslash)
“ (double quote)
` (backtick)

.. in your testcases or test strings.
These characters will cause things to happen either at the terminal or in your code that will be difficult to deal with.

3. Examples

./tokenizer "today is a beautiful day"
word: “today”
word: “is”
word: “a”
word: “beautiful”
word: “day”

./tokenizer "0x4356abdc 0777 [] "
hexadecimal integer: "0x4356abdc"
octal integer: "0777"
left bracket: "["
right bracket: "]"

4. Extra Credit

In C the keywords like if, while, etc are distinct tokens that a compiler recognizes separate from the identifiers used for variable names, function names and the like. One opportunity for extra credit is for your tokenizer to recognize the C keywords as distinct tokens.

Your second opportunity for extra credit is to recognize and skip C comments. Comments are not tokens and should be treated like white space. In C, there are two forms of comment.

A third opportunity for extra credit is to recognize strings in double or single quotes as single tokens. So at the least, any sequence of characters (including blanks and tabs) would be considered to be one token.