CS 214: Systems Programming, Spring 2015
Programming Assignment 1: A Real Tokenizer

1 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, although you will be dealing with some pointers to structs as well.

Your task is to write a type and a set of functions equivalent of a Java class that implements a tokenizer. The tokenizer should accept a string as a command-line argument. The string will contain one or more *tokens* separated by *white space*. White space is defined as any sequence of blank (0x20), tab (0x09), new-line (0x0a) and carriage return (0x0d) characters. The tokenizer should return the tokens in the string one token at a time, hence your program is called a tokenizer.

The complexity of this assignment means that you will have to plan out the behavior of your program before you write any code. You can plan out the behavior by writing a *Finite State Machine* for your tokenizer. A finite state machine is essentially a transition diagram drawn as a graph. The nodes in the graph are states and the arcs are transitions from one state to another. Labelled arcs imply transitions associated with the characters used as labels. The machine is in only one state at a time. The state it is in at any given time is called the current state.

There may be characters in the input that are neither part of tokens nor white space. Many of these special characters are either unprintable or have undesirable effects on program output. These special characters are called *escape characters*. Characters that are not part of any token or white space should be output in error messages from your program. In your output, we want the output of all escape characters (printable and otherwise) to be in bracketed hex of the form [0xhh]. So if the command line input string contains a vertical tab (0x0b), your error message would represent the vertical tab as "[0x0b]".

2 Implementation

Your implementation needs to export the interface given in the attached `tokenizer.c` file. In particular, you need to define the type needed to represent a tokenizer and three functions for creating and destroying tokenizer objects and getting the next token. Note that we have only defined the minimal interface needed for external code (e.g., our testing code) to use your tokenizer. You will likely need to design and implement additional types and functions.

A token is a sequence of any ASCII characters that does not contain a white space character. Tokens are separated by one or more white space characters. Multiple white space characters may be next to each other (see second example above), and/or at the beginning and/or end of the token string. When this happens, your tokenizer should discard *all* white space characters.

There are different kinds of tokens. Your program must not only break the input string into tokens, it must identify the *kind* of token in the program output.

A *word* token is an alphabetic character followed by any number of alphanumeric characters.

A *decimal integer constant* token is a digit (0-9) followed by any number of digits.

An *octal integer constant* token is a 0 followed by any number of octal digits (i.e. 0-7).

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

A *floating-point constant* token is follows the rules for floating-point constants in Java or C.

C operator tokens are any of the C operators shown in your C language reference card.

Your implementation *must not* modify the original string in any way. Further, your implementation must return each token as a C string in a character array of the exact right length. For example, the token usr should be returned in a character array with 4 elements (the last holds the character '\0' to signify the end of a C string).

You may use string functions from the standard C library accessible through string.h (e.g, `strlen()`).

You should also implement a `main()` function that takes a string argument, as defined above. The string contains zero or more tokens separated by white space characters. Your `main()` function should print out all the tokens in the argument string in left-to-right order. Each token should be printed on a separate line. Here is an example invocation of the tokenizer and its output.

tokenizer " array[xyz] += pi 3.14159e-10 "

word "array"

left brace "["

word "xyz"

right brace "]"

plusequals "+="

word "pi"

float "3.14159e-10"

Looking at the C reference card (available at the class website), 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.

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 minimal number of arguments needed.

Your code should compile correctly (no warnings and errors) with the `-Wall` and either the `-g` or `-O` flags. For example

```
$ gcc -Wall -g -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: You may write your code on any machine and operating system you desire, but the code you turn in MUST (un)tar (see below), compile and execute on the iLab machines or a zero grade will be given. Be sure to compile and execute your code on an Ilab machine before handing it in. This has been clearly stated here and NO EXCEPTIONS will be given.

3 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 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 would be 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.

4 Examples

4.1 Basic Input

- Input:

```
./tokenizer "today is a beautiful day"
```

- Output

```
word "today"
word "is"
word "a"
word "beautiful"
word "day"
```

4.2 Multiple Delimiters

- Input:

```
./tokenizer "0x4356abdc 0777 [] "
```

- Output

```
hex constant "0x4356abdc"
octal constant "0777"
leftbrace "["
rightbrace "]"
```

5 What to turn in

A tarred gzipped file named pa1.tgz that contains a directory called pa1 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.

Suppose that you have a directory called `pa1` in your account (on the iLab machine(s)), containing the above required files. Here is how you create the required tar file. (The `ls` commands are just to help show you where you should be in relation to `pa1`. The only necessary command is the tar command.)

$ ls pa1 $ ls pa1 readme.pdf testcases.txt tokenizer.c $ tar cfz pa1.tgz pa1

You can check your `pa1.tgz` by either untarring it or running `tar tfz pa1.tgz` (see `man tar`).

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).