CS211 Computer Architecture

Project1: Basic Tokenizer
Due on: 11:59PM Wednesday July.13rd

1 Introduction

In this assignment, you will practice programming with C pointers. Much of the pointer manip- ulation 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 similar to a Java class that implements a tokenizer for C language numeric constants. The tokenizer should accept a string as a command-line argument. The string will contain a file name, in which there are one or more tokens separated by blank space or a line-feed. Each token is a either a floating-point constant, or an integer constant in hex, decimal or octal. The later definition of tokens will define what begins each kind of token and how the tokens end.

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. These 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 these characters to be in bracketed hex of the form [0xhh]. So if the file contains a string with a comma ’,’(0x2c), your error message would represent the comma as ”[0x2c]”.

Parsing result should be printed to a file called ”result”. Each token stays in a separate line. Error message should be printed to another file called ”error.msg”. Each hex number stays in a separate line.

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 as you needed.

Tokens may be separated by white space characters. Multiple white space characters may be next to each other, and/or at the beginning and/or end of the token string. When this happens, your tokenizer should discard all white space characters.

A *decimal integer constant* token is a digit (1-9) 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 follows the rules for floating-point constants in Java or C.

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 end-of-C-string character).

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 file name argument, as defined above. Your main() function should print out all the tokens in the file in left-to-right order. Here is an example invocation of the tokenizer and its output.

file named ”tokens”

0700 1234 3.1415e-10 ,

Call line of your executable:

./tokenizer [path to file dir]tokens

file created named ”result”

octal ”0700”

decimal ”1234”

float ”3.1415e-10”

file created named ”error.msg”

[0x2c]

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.

3 Steps

There could be more than 100 ways to finish the project, here’s just one I recommend.

3.1 Compile the dummy code structure

You are given 3 program files:

”main.c”, ”tokenizer.h”, ”tokenizer.c”

”tokenizer.h” contains the structure and helper functions declarations, which will be used by ”main.c”

”tokenizer.c” contains the implementation of all functions in ”tokenizer.h”

”main.c” contains the main flow of the program.(accept file name and get result back from tokenizer and write to file)

Write a **makefile** that would link all 3 files together.

Note: there’s a dummy function being provided in ”tokenizer.h”. Your ”main.c” should be able to call that function. In other world, you should be able to compile the whole

project when you type ”make” at the command line. And see ”hello world” being printed out to terminal when you run the program.

3.2 Implement the ’lexer’

A lexer should be able to filter out all the blank spaces and line-feeds in the file and store all the tokens(no matter what types are they, or even right or wrong) in a data structure. For example, in the previous example, your lexer should be able to create an array of length 4. The element in the array would be char pointers of ”0700”, ”1234”, ”3.14159e-10”, and ”,”.

Place the pointers to tokens in an array named ’tokenArray’. tokenArray[]:
{”0700”, ”1234”, ”3.14159e-10”, ”,”}

3.3 Implement the ’parser’

A parser should be able to interpret the string one at a time. Your parser would iterate through the array of tokens and determin their types or whether they are correct or not.

Place the recognized result in an array named ’result’. result[]:
{”octal”, ”decimal”, ”float”, ”[0x2c]”}

3.4 Implement the ’writer’

A writer should be able to iterate through the result array and print everything to files. e.g. if it’s ”octal”, print ”octal” and the corresponding token in the same index in tokenArray[] ”0700” to a line in file ”result”

4 Special Cases

You can assume the input always follow the rules below:
1. All tokens will be separated by white spaces(1 or more)
2. Each token would either be a valid constant number, or a single invalid character
3. Each line would have maximum length of 50 characters
e.g. You won’t need to consider the following conditions
”123,456” ”078e-10”
”123456” will be treated as Decimal 123456, rather than 2 Decimals: 123 / 456

5 What to turn in

IMPORTANT NOTE: You may write your code on any machine and operating system you desire, but the code you turn in MUST be able to compile and run on ILAB or a zero grade will be given, since our TAs will grade your work on ILAB. 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.

A tar ball named pa1.tar.gz shall be submitted with at least 5 files: - main.c - tokenizer.c - tokenizer.h - makefile - readme.pdf, it contains a brief description of the program

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 ball. (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
$ tar -zcvf pa1.tar.gz pa1
```

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