Assignment 1

This program is designed to test your proficiency in C or C++ and demonstrate that you have the prerequisite knowledge in one of these two languages. This assignment must be done by yourself; teams are not allowed. An automated program structure comparison algorithm will be used to detect software plagiarism.

While a human will inspect your program for style, correct output will be judged by a computer and your output must match perfectly. Failure to match the output in this assignment by the due date posted on the course calendar will result in your being removed from the class for lack of prerequisites. If you are interested in joining the class, you must complete this assignment by the due date.

This program must run in your account on edoras.sdsu.edu. If you are enrolled in the class, you will have already been e-mailed an edoras account. If not, you should sign up for one in class.

Operating systems communicate with users via a *shell* program that reads input (e.g. voice, keyboard, mouse) and follow user directives to launch or otherwise control programs. In this assignment, you will write a small program that processes keyboard input and would be the first step in a shell program. To make things simpler, you will use the GNU readline library which reads a line of input from a user and allows editing of the line.

To use readline, you must include stdio.h prior to including readline/readline.h. When you compile the program, you must use the link library (-l) flag after the file(s) that you are compiling. In this assignment, all code should go in file tokenizer.c (C) tokenizer.cpp (C++). You should compile to program tokenizer. Example:

```
# Note: no space between -l and the library name
#       -l after the targets to compile
g++ -o tokenizer -g tokenizer.cpp -lreadline
```

You may wish to use gcc if you are writing in C. You will need to create a Makefile to do this automatically (see below). Your tokenizer program is responsible for doing the following:

Print a command prompt "> " and read a line of data using function readline (`char * readline(char *prompt)`). In general, more detail about library functions can be had by typing "man functionname" at a UNIX prompt. Take the string that is returned and from it create a linked list of tokens. Each token should be in a structure that has a character pointer to the token value and a next pointer that is either NULL or points to the next token. Your program will then print the contents of the linked

list in square [ ] brackets. The token values will be surrounded by curly braces { } and separated by a comma. See below for examples.

The rules for tokens are as follows and are loosely based on rules used by many UNIX shells:

1. Whitespace (space, tab, etc) ends a token and is skipped unless it is in quotes (see below). You will find the function `int isspace(char)` useful for detecting whitespace characters.
2. Several characters are considered special tokens: | ; < > &
   When one of these characters is encountered, the token being built is completed and a new token consisting of the character is created.
3. The special character \ is an escape character. The next character will be part (or the start) of a token. For example, normally the string "now;is" would be three tokens: `[{now},{;},{is}]`. However, "now\;is" results in a single token: `[{now;is}]`.
4. Items between single or double quotes are treated as parts of the current token:

   ```
   > me gusta UNIX  → [{me},{gusta},{UNIX}] vs.
   > me" gusta UNIX" → [{me gusta UNIX}]
   ```

   Quoted text can only be terminated by the same type of quote.
   The escape character only works inside double quotes.

   ```
   > So long "and thanks's for the fishes"
   [{So},{long},{and thanks's for the fishes}]
   ```

   For simplicity, quotes are automatically terminated when a line is processed if the user did not do so. Most shells will usually allow multiline entry across quotes or the escape character, but we will not do this. DO NOT GET AMBITIOUS AND IMPLEMENT THIS, you will lose points.

5. Characters that do not meet the above rules are added on to the current token being built.

Examples:

```
> let\'s do some "crazy \"air quotes\""
[{let's},{do},{some},{crazy "air quotes"}]

> ls | sort >sorted_files.txt
[{ls},{|},{sort},{>},{sorted_files.txt}]

> cat -n /etc/fstab
[{cat},{-n},{/etc/fstab}]
```

```
> climate --reverse="50 years"&
[{climate},{--reverse=50 years},{&}]
```

You will also need to do the following that are *critical* to the grading algorithm being able to grade your program:

- Use an editor to a .plan file in your home account. This is a file that is used with the finger command, which provides information about users, e.g. finger mroch. It should contain the following, updated with your information:

  ```
  Student: Kikumba, Jules
  Nickname: Papa Wemba  (you may omit this line)
  RedID: 123 45 6789
  ```

- Your program must have a Makefile. Makefiles are program compilation specifications. The make tool examines to see if source files have changed and will compile only when sources are newer than executables. See the class FAQ for details on how to create a makefile. You can test if your makefile works by typing make. The makefile should produce a program called tokenizer.

- The grading program will look for your solution in directory cs570/a01 relative to your home directory. You must place it there.

- The comparison program will only look at lines of output that start with an open square bracket.

The comparison program will be run from time to time and will leave a file called autograder.txt in your home directory. It will contain a timestamp letting you know when it was run and the results. **Start early on this assignment, your enrollment in this class depends on it.**

Be sure to comment your code appropriately. Code with no or minimal comments are automatically lowered one grade category.

What to turn in:

- Hardcopy of your code with sample output.
- Affidavit of academic honesty (see FAQ)