Assignment 5: Shell, pt. 1

Due: Thursday, February 24, 10pm

Starter code: See Assignment 5 on Canvas for the Github Classroom link.

Submission: This is a **pair assignment**, but you can work alone, if you so choose.

Submit the contents of your repository via Gradescope. See Deliverables below for what to submit. If you are working with a partner, do not forget to include their name with the submission.

This a assignment if the first of two parts. The first part will ask you to build a basic shell that is capable of running user-specified commands.

Task 1: Shell Tokenizer

Before we can execute commands (or combination of them) we need to be able to process a command line and split it into chunks (lexical units), called *tokens*. The input of a tokenizer is a string and the output is a list of tokens. Our shell will use the tokens described in the table below. The tokens `(`, `)`, `<`, `>`, `;`, `|`, and the whitespace characters (space `' '`, tab `'\t'`) are *special*. Whitespace is not a token, but might separate tokens.

Token(s)	Description / Meaning
`( )`	Parentheses allow grouping shell expressions
`<`	Input redirection
`>`	Output redirection
`;`	Sequencing
`|`	Pipe
`"hello (world;"`	Quotes suspend the meaning of special characters (spaces, parentheses, ...)
`ls`	Word (a sequence of non-special characters)

Your first task is to write a function that takes a string (i.e., `char *` in C) as an argument and returns a list (array, linked list, etc.) of tokens. The maximum input string length can be explicitly bounded, but needs to be at least 255 characters.

You also need to provide a demo driver, `tokenize.c` that will showcase your function. The driver should read a single line from standard input and print out all the tokens in the line, one token per line. For example:

```
$ echo 'this < is > a demo "This is a sentence" ; "some ( special > chars"' | ./tokenize
this
<
is
```

```
>
a
demo
This is a sentence
;
some ( special > chars
```

In this example, we print the example string to standard output, but immediately *pipe* that output into the input of the `tokenize` program. We will implement piping in our own shell in the next assignment.

Whitespace that is not in quotes should not be included in any token.

To help you get started writing a tokenizer, see the example included with the starter code.

What we are implementing here is a Deterministic Finite-state Automaton (DFA), which is a recognizer for Reguar languages. While not necessary to complete this assignment, you might want to read up on those to get a deeper understanding if you are interested.

Task 2: Basic Shell

The second task is to implement basic shell functionality running a single user-specified command at a time. The shell should display a prompt, read the command and its arguments, and execute the command. This should be performed in a loop, until the user requests to exit the shell. The commands can have 0 or more arguments and these arguments may be enclosed in double quotes `"`, in which case the enclosed string is treated as a single argument.

Example interaction:

```
$ ./shell
Welcome to mini-shell.
shell $ whoami
ferd
shell $ ls -aF
./        .git/     shell*    shell.o   tokens.h  vect.c    vect.o
../       Makefile  shell.c   tokens.c  tokens.o  vect.h
shell $ echo this should be printed
this should be printed
shell $ exit
Bye bye.
```

Here are the requirements for the basic shell

1. After starting, the shell should print a welcome message: `Welcome to mini-shell.`
2. You must have a prompt `shell $` in front of each command line that is entered.
3. The maximum size of a single line shall be at least 255 characters. Specify this number as a (global) constant.

4. Each command can have 0 or more arguments.
5. Any string enclosed in double quotes (`"`) shall be treated as a single argument, regardless of whether it contains spaces or special characters.
6. When you launch a new child process from your shell, the child process should run in the foreground by default until it is completed. The prompt should be printed again and the shell should wait for the next line of input.
7. If the user enters the command `exit`, the shell should print out `Bye bye.` and exit.
8. If the user presses *Ctrl-D* (end-of-file), the shell should exit in the same manner as above.
9. If a command is not found, your shell should print out an error message and resume execution.

   For example:

   ```
   shell $ ano;wavu;
   No such file or directory
   shell $
   ```

10. System commands should not need a full path specification to run in the shell.

    For example, both `ls` and `/bin/ls` should work.

Using the Provided Makefile

As before, we provide you with a Makefile for convenience. It contains the following targets:

- `make all` – compile everything
- `make tokenize` – compile the tokenizer demo
- `make tokenize-tests` – run a few tests against the tokenizer
- `make shell` – compile the shell
- `make shell-tests` – run a few tests against the shell
- `make test` – compile and run all the tests
- `make clean` – perform a minimal clean-up of the source tree

Deliverables

Task 1 Implement `tokenize.c`. Implement the tokenize function in a separate `.c` file and also provide a header file so you can use it in Task 2.

Task 2 Implement the basic shell in `shell.c`.

For each task, include any `.c` and `.h` files your implementation depends on and commit everything to your repository. Do not include any executables, `.o` files, or other binary, temporary, or hidden files.

Finally, go to Gradescope and submit your repository (or a ZIP archive which can be downloaded from Github). This step is required – committing to Github is not enough. Make sure you include your partner in the submission.

Hints & Tips

- The starter code repo contains an example of a tokenizer for a simple arithmetic language containing integers and the operations `+`, `-`, `*`, and `/`. Use this example to help you get started on the first task.
- Use the function `fgets` or `getline` to get a line from `stdin`. Pay attention to the maximum number of characters you are able to read. Avoid `gets`.
- Figure out how `fgets/getline` lets you know when the shell receives an end-of-file.
- Use the provided unit tests as a minimum sanity check for your implementation. Especially before the autograder becomes available.
- **Write your own (unit) tests.** Doing so will save you time in the long run, especially in conjunction with the debugger. In office hours, the instructors or the TAs may ask you to show how you tested code that fails.
- Follow good coding practices. Make sure your function prototypes (signatures) are correct and alway provide purpose statements. Add comments where appropriate to document your thinking, although strive to write self-documenting code.
- Split code into short functions. Avoid producing "spaghetti code". A mutli-branch **`if-else if-else`** or a multi-case **`switch`** should be the only reason to go beyond 30-40 lines per function. Even so, the body of each branch/case should be at most 3-5 lines long.
- Use `valgrind` with `--leak-check=full` to check you are managing memory properly.
- A string vector implementation might be useful.
- Avoid printing extra lines (empty or non-empty) beyond what is required above. This goes both for the tokenizer and the shell. Extra output will most likely confuse our tests and give false negatives.

Changelog

02/18/2022

- Task 1: Explicitly state that the input size can be bounded.
- Hints & Tips: Suggest `getline` in addition to `fgets`.