Assignment #2

CS 180 Fall 2020

Deadline:	Part 1: As specified on the moodle
Topics covered:	Process creation and management
Deliverables:	To submit all relevant files that will implement the uShell program and provide a Makefile. Please zip up the file according to the naming conventions laid out in the cs180 syllabus. Your program must be compile-able in the Cygwin environment using the `/usr/bin/gcc or g++` compiler.
Objectives:	To learn and understand the basic of the internals of a shell program.

Programming Statement: A new shell program called uShell

In this assignment, you are tasked with the assignment to create a basic shell program called **uShell**. A shell program behaves typically in a similar fashion to a command prompt. Usually, it has the following capabilities:

- Ignore comments statement
- Perform internal commands (Part 1)
- Run programs and launch processes (Part 1)
- Setting of environment variables (Part 1)
- Perform input/output redirection (To be done in Part 2)
- Performing piping as a means of interprocess communications between processes (To be done in Part 2)

1 An overview of uShell

Your program will be referred to as `uShell.exe`. It is to be run either in interactive mode or as a shell script. Thus, the input to `uShell.exe` can either be from the terminal or a file.

1.1 Usage

`uShell` is to be run as follows:

```
uShell.exe [-v]
```

The bracketed arguments are optional. Their meaning are as the following:

- If the `-v` argument is used, the Shell is running in the verbose mode. In the verbose mode, every command line entered is repeated.

Some examples:

- `uShell.exe` – No arguments at all. Input from terminal.
- `uShell.exe -v` – Input from terminal, but every line entered is printed before execution.

1.2 Input

The program reads a command line one at a time from the standard input until the end of file or when a `exit` command is encountered. Every command line is terminated upon a newline character. Every command line is either empty or a sequence of *word*, where every *word* is a character sequence of printable characters but not whitespace characters. The length of the command line is usually limited by the parameter *ARG_MAX* defined by Cygwin, e.g. 32000.

Hence, it is possible for a command line to be empty – it is simply a newline character or it is made up of only whitespaces. Furthermore, there is a special category of command line that can be ignored by the program – a comment. At any point in a command line, if the `#` sign is used, the rest of the line is treated as a comment and it can be ignored by the shell program.

1.3 Command line examples

The following shows some example of possible command lines. As you can see, there are both what is termed internal commands and external commands. The difference between the two will be made clear in later sections.

Command	Explanation
`# comment haha`	*This is a comment*
`/usr/bin/gcc.exe`	*An external command - executing a program with full path given*
`ls.exe -l`	*An external command - executing a program to be found among the directory list in environment variable PATH*
`echo hello`	*An internal command - printing the word hello to standard output*
`echo hello # silly comment`	*An internal command - printing the word hello to standard output.* `silly comment` *is ignored because it occurs after the* `#` *symbol*

When standard input is used as input, there should be a prompt printed before accepting input. For example,

```
uShell> echo haha # whatever
haha
uShell> ..
```

As explained in the next section, the default prompt is `uShell` but it can be changed by the `changeprompt` internal command.

1.4 Internal Commands

An internal command is directly executed by the shell itself without executing another program. The first word in the command determines which internal is to be executed. The internal commands are:

- `echo`: print out the rest of the arguments.
- `exit <num>`: terminate the shell program with an exit value given by `num`. If `num` is not given, the default exit value is 0.
- `changeprompt <str>`: change the prompt to the string given by `str`.
- `setvar`: described in Section 1.5 on shell variables.

1.5 Shell Variables

Any word in a command line that begins with the character `$` and enclosed with curly braces refers to a shell variable. The default value of a variable is an empty string i.e., if a shell variable is used without any initialization, it is an empty string. The value can be changed via command line. The command to define and change the value of a variable is `setvar`. The usual syntax of the `setvar` command is as follows:

```
setvar <varname> <value>
```

The names of the shell variable are case-sensitive. To call out a shell variable, the '`$`' character is used to indicate that a variable is used. The following examples demonstrate the usage of the `setvar` command.

```
uShell> setvar HAHA hoohoo # assign the value hoohoo to HAHA
uShell> echo ${HAHA} # calling out the value of HAHA
hoohoo
uShell> # hoohoo is printed on the screen.
uShell> setvar haha # variable haha is defined and given a default value.
uShell> echo ${Haha}123 # Attempting to call out the value of an undefined variable.
Error: Haha123 is not a defined variable.
uShell> echo ${HAHA}123 # disambiguate the beginning and end of the var name
hoohoo123
uShell> echo ${HAHA }123 # wrong use of curly braces. Would be read as 2 separate words.
${HAHA }123
uShell> echo $${HAHA} # $ sign can be used together with variables
$hoohoo
uShell> echo ${${HAHA}} # nested use of curly braces are  not supported
${hoohoo}
uShell> # So the replacement only happen once.
```

Please ensure that your program prints out the same behaviour as given above.

1.6 External Commands

An external command is a sequence of words where the first word does not match any of the internal commands given above. There are two forms of external command. They are:

- **Pathname commands**: where the first word will contain a '`/`' character. Example: `/usr/bin/gcc.exe`. It must begin with a '`/`' character. We will only support what is called absolute pathnames.

- **Filename commands:** where the first word does not contain any '/' character. Example: `ls.exe -l`.

For the external commands where the filenames only are given, the executable file is to be found in the list of directories given in the `PATH` variable. The `PATH` shell variable is a special variable that will be used in the context of the external commands. The `PATH` variable is a variable that is a sequence of pathnames separated by a colon character '`:`'. `PATH` variable is initialized from the PATH environment variable in Cygwin.

So if the `PATH` variable is "`/bin:/usr/bin"`, there are two directories given – `/bin` and `/usr/bin`. So the shell program searches through the directories in the `PATH` variable to find the executable for running the external command. If the executable with the given filename is not found in all the directories or the given absolute path executable cannot be found, the shell program is supposed behave as the following example shows:

```
uShell> foo/goo/bin.exe haha # not using an absolute path.
Error: foo/goo/bin.exe cannot be found
uShell> /foo/goo/bin.exe haha # example using non-existent path or filename
Error: /foo/goo/bin.exe cannot be found
uShell> bin.exe # bin.exe cannot be found in any directories in PATH variable
Error: bin.exe cannot be found
```

For the sake of ease, we ask that the user must provide the full filename and not merely filenames without the proper extensions. For example,

```
uShell> ls haha
Error: ls cannot be found
uShell> ls.exe haha # haha is a file in current directory
haha
```

Normally, the external commands must be executed in a child process of the shell program and the shell program will immediately wait for the child process to finish before continuing execution.

2 Rubrics

The total marks for this assignment is 100. The rubrics are as the following:

- (5 marks) Process comments. Your `uShell` should be able to ignore anything that comes after the `#` character. But `#` should not be part of a word. So `echo dff#sea` prints `dff#sea`.
- (5 marks) `exit` command. Your `uShell` should be able to exit with the correct exit status value that is given from the input command line.
- (5 marks) `changeprompt` command. Your `uShell` should be able to change the prompt string when the user used the `changeprompt` command.
- (30 marks) Support shell variables. Your `uShell` should be able to support the setting of the Shell variables as described in section 1.5 above. In particular, should the user attempt to

- Call out an existing variable properly e.g., ${HAHA}123, abc${Haha}def, the part of the command line must be replaced by the value of the existing variable.

- Call out a non-existing variable properly e.g., ${HoHo}123, abc${Hoho}def, but these variables have not been previously set with a `setvar` command. `uShell` should print out a suitable error message.

- Use whitespace within the curly braces e.g., ${HAHA }, ${HA HA}, your `uShell` must recognize them as two separate words.

- Attempts of nested use of calling out Shell variables are not supported. Hence only 1 replacement should happen for each of these occurences.

- (10 marks) `echo` command. This `echo` command should be able to re-print the rest of the command line properly. Multiple whitespaces may be ignored and treated as if it is a single whitespace.
- (30 marks) External commands. You are expected to support both pathname and filename commands. If your `uShell` cannot support Shell variables and hence cannot support filename commands, you will only get 15 marks for this part of the rubrics.
- (5 marks) Comments. Comments ought not to be verbose, but explain the big picture and use good variable names to indicate. Readability of your code is key here.
- (10 marks) Structure of your code. Avoid hard-coding. If your code is not easily re-usable, contain things such as unnecessary cascading if statements, you will lose marks here. For example, a polymorphic design for the Command objects would be a nice touch.
- The only language allowed for this should be C or C++. You are expected to provide a Makefile that should work i.e., the following command

  ```
  make uShell
  ```

  should create a `uShell.exe` succesfully in the Cygwin environment found in your lab without errors or warnings.