For this assignment, we will continue work on the micro-shell. Start with the code base you have for the last assignment, a2. Continue using branch master for your work. Make sure you don’t work on the a2 branch.

This assignment’s work consists of the following steps:

- Correct any problems you know about from Assignment 2 if you know how. Ask for help if you don’t know how to fix the problems.

- (10 points) Add comments to your shell. All characters on an input line starting at and after a pound sign (#) are comments. A special case is $# which is not the start of a comment. It must continue to be processed as a standard variable. This should be done before in main() before the call to processline(). It may be done by or from a helper function. It should be done at the same time as removing the newline, that is, you remove a comment or the newline if a comment is not there.

- (100 points) Add argument processing for arguments to ush’s main. This is not argument processing for arguments produced by arg_parse, it is for the arguments passed to your shell when your shell starts. (The parameters to ush’s “int main (int mainargc, char **mainargv) ...”. The name mainargc and mainargv could be any name, I just named them that to show you that they are the argc and argv that are parameters to main() for your shell.) This would be the arguments used when your shell is run. For example, your ush could be run like:

  ush filename arg1 arg2 arg3 arg4 arg5

instead of

  ush

To add argument processing for main’s arguments, do the following:

- If the shell is started with one command line argument (the shell’s name), all input comes from the standard input. This is considered an interactive execution. (This is what your ush currently does. You start your ush by giving the command “ush” or “./ush”.)
If the shell is started with more than one command line argument, shell commands are read from the file named in the first argument after the shell’s name. (The first argument is the script file name. For example, you would start ush running with the command 
"./ush script arg1 arg2" and ush would read commands from the file named “script”.) If ush can not open the file for reading, ush should print an error message and exit with the value of 127. When ush has read all lines from the script file, it should exit with the value 0.

Note: Again, this is how you start ush by giving command line arguments to the shell that starts ush. You will need to use mainargc and mainargv as passed into the main() function of your ush program.

Do not print a prompt when reading from a script file. An interactive run still must print a prompt to standard error.

Lines from either standard input or from the file are processed exactly the same way. They are sent to “processline()” for processing. (In fact, the only change in the main loop should be which file is read by fgets(). The best way would be to have only one fgets(). Remember, variables can vary and you can have stream variables. (Type FILE *.))

Add processing to “expand()” that recognizes the pattern $n, where n is an integer greater than or equal to 0. “expand()” should replace the $n by the the shell’s command line argument n+1. (Yes, n may be a multi-digit integer.) When run with multiple command line arguments, $0 should be replaced by the script name. If n is larger than the number of command line arguments, $0 should be replaced with the empty string. (The empty string really means that the $n pattern in the command is just removed from the command.) When the shell is run without any arguments, that is during an interactive execution, $0 should be replaced with the name of the shell. All other $n values are replaced by the empty string.

“expand()” should also replace $# by the base 10 ascii representation of the number of arguments. That is, the number of arguments indexed from $0 to $n-1 where n is the value of $#.

For example, if your ush was run using the command:

```
ush script_name a b c d
```
$0$ should be replaced by “script_name”, $1$ by “a” and so forth. Also $\#$ should be replaced by 5. (Note: the argc parameter to ush’s main() function would be 6.)

– Implement a new built-in command with the syntax:

\[
\text{shift } [n]
\]

that shifts all arguments starting with $1$ by $n$ values. That is, parameter $i+n$ now becomes parameter $i$ and for $i = 1$ to $\#-n$. Also, $\#$ is also decremented by $n$. If $n$ is not given, $n$ is 1. Note, $0$ is still the script name and $\#$ should never be less than 1. If a shift command requests a shift that can not be done, that is $n \geq \#$, give an error message and do not shift the arguments. Note, arguments $1$ through $n$ are no longer available to use in $i$ replacement.

– Implement a new built-in command with the syntax:

\[
\text{unshift } [n]
\]

that does the opposite of shift. If $n$ is not specified, unshift should completely undo any effects of all earlier shift commands. If $n$ is specified, unshift the number of arguments specified by $n$. If $n$ is greater than the number of shifted arguments, return an error message and do not shift the arguments.

As an example, assume you have a file called “showshift” that contains the following commands:

```
  echo showshift is named $0
  echo Number of arguments is $\#.
  echo Argument 1 is $1.
  echo Argument 2 is $2.
  echo Argument 3 is $3.
  echo Argument 4 is $4.
  shift 3
  echo Number of arguments is $\#.
  echo Argument 1 is $1.
  echo Argument 2 is $2.
  echo Argument 3 is $3.
  echo Argument 4 is $4.
  unshift 1
  echo Number of arguments is $\#.
  echo Argument 1 is $1.
  unshift
```
echo Number of arguments is $#.
echo Now $1 is Argument 1.

Now assume you give the command “ush showshift a b c d e f”.
The output should be:

    showshift is named showshift
    Number of arguments is 7.
    Argument 1 is a.
    Argument 2 is b.
    Argument 3 is c.
    Argument 4 is d.
    Number of arguments is 4.
    Argument 1 is d.
    Argument 2 is e.
    Argument 3 is f.
    Argument 4 is .
    Number of arguments is 5.
    Argument 1 is c.
    Number of arguments is 7.
    Now a is Argument 1.

Note: Your implementations of shift and unshift must not change the mainargc or mainargv variables or any copy of them.

Note: While the above script has the $n variables near the end of the line, they may appear anywhere in the line as is shown on the last line and there may more than one replacement needed. You can get a copy of the showshift script the usual way from the class website.

• (35 points) Add a limited wildcard expansion capability to expand(). The wildcard character for your shell is ‘*’. In the simple version, the ‘*’ appears by itself. White space (or the beginning or end of the line) must be on either side of the ‘*’. In this case, replace the ‘*’ by names of all the files in the current working directory who’s name do not start with a period. In the more complex version, if the ‘*’ appears with a leading white space and trailing non-white space, that is something like “ *xyz ”, replace the ‘*’ and the following characters with all file names in the current directory that end with the context characters. The context is terminated by white space or end of string. Next, if the line does not match either of the two cases above, just copy the ‘*’ to the expanded string. Finally, the sequence ‘\*’ should put only
the '*' in the expanded string. (Again, do not use file names that start with a period.) The file names placed into the expanded string must be separated by a single space character between file names. You should not have leading or trailing spaces. Note: If the trailing context characters include a slash (/), generate an error message and stop the processing of the current line. Also note: If no file names are matched, the pattern must be put in the expanded string. Finally, don’t use a regular expression matching library to implement wildcard expansion. You are to use opendir(3), readdir(3) and closedir(3). Do not use scandir(3).

• (30 points) Add a new built-in command “sstat file [file ...]” that prints the information from stat(2) system call for each file on the command line. Your output for each file must be as follows:
  – the file name
  – the user name (not the uid)
  – the group name (not the gid)
  – the permission list including file type as printed by ls(1).
  – the number of links
  – the size in bytes
  – the modification time (use localtime(3) andasctime(3))

These fields should be separated by a single space between the fields. Each file should be listed on a separate line. If there is no associated user name or group name, print the uid or gid in numerical form. To help converting the stat(2) information to a printable form, you may use the file /home/phil/public/csci347/strmode.c. “man strmode” will help you understand the file. Add it to your project if you choose to use it. You will not need to use the “-lbsd” flag to link.

• The remaining 25 points are for style, turn in points, following directions and so forth.

• Add no other features.

• If you add a new file to your repository, don’t forget the check it in and push it. Also, don’t forget the style guidelines.

• Again, I will grade your assignment by checking out your sources, running make and then testing your shell.
• When you have completed this assignment and are ready to turn it in, at that point, create branch 'a3' and push it to origin. (If you forget how to push a new branch, the assignment 1 write up has the instructions.)

• What to turn in: Turn in your source code and some tests of your own to show your shell is working. Please make sure you include the source of your ush testing scripts along with the output. Also, include a run of my test script with your output.

Notes:
I'd like to give you a few pointers. I believe these can reduce the time required to do this assignment.

• Do things as simply as possible.

• gdb can help reduce debugging time. If you have not watched the video on gdb which is linked from the class web site, you should watch it. Single step your program through the parts of your program you are developing.

• Have only one main loop. Figure out a way to get your line from the proper source instead of having a loop for script processing and another loop for interactive processing.

• Continue having processline() your primary function for running commands. processline() should not care whether you are doing an interactive run or are processing a script.

• Use global variables if information is required in many places. C tends to need global variables more than other languages. Don’t make all variables global, but do use globals if it is the easiest way to get data where it is needed. For example, it might be useful to have global variables for access to mainargv and mainargc from the main function. main() can initialize the global variables from its parameters. (See /home/phil/public/csci347/global for example files of one way to use globals.)

• The library function assert(3) can help you find bugs in your logic. Read about it in the man pages.

• Add one feature at a time and get it working. For example, comment removal, then processing of commands from a script file, then $n, $#,
shift and unshift, sstat, and finally your limited wildcard processing. *When you get a feature working check-in your working sources.* Your commit messages should describe the changes, not just say something like “changed ush.c”. That is obvious from your commit, have your commit message be more like “ush.c: added code to read from a script file”.

• hint for $n implementation: man 3 isdigit

• If you want to run the testa2 scripts on assignment 3, you can cd to the testa2 directory and run “try” with the switch ‘-M’. (Make sure it is ‘M’, not ‘m’.) Your run should look like “./try -M” and that asks the script to use master rather than the branch it normally wants to use. If you use the -H flag, it must be first. e.g. “./try -M -H”

• There will be a testa3 directory that will be available around Thursday July 6, 2023. If it is not available after that time and you need access, send me an e-mail asking for access.