Start with the code you turned in for assignment 1. You should have a single file “ush.c” in a “ush” directory which has three primary functions, arg_parse(), processline() and main(). (You may have helper functions for arg_parse().) Before you start work on assignment 2, make sure your git project is on the “main” branch. “git checkout main” is the command to make sure you are on the main branch.

This assignment’s work consists of the following steps:

• (20 points) Add a new file, “Makefile”, that makes the executable “ush” from your ush.c file. Just giving the command “make” in your directory should make the program. You should also have a “clean” target that removes all generated files including the final executable. You may need to read about the “make” program to get this working the way you want it.

The following file is a start:

```bash
# This is a very simple makefile
#
CFLAGS=-g -Wall
ush: ush.o
    $(CC) $(CFLAGS) -o ush ush.o
```

The “$(CC)” line must start with the tab character, not spaces.

As this project continues, you will need to add some new .c or .h files to your project. When you add the .c file to your project, you need to update the Makefile so it compiles your ush from all the .c files. When a .h file is added, you may need to add dependency lines.

When you upgrade your Makefile it must do the following:

– compiles each .c file to a .o file in separate steps.
– links all .o files together into the ush executable separate from compiles.
– uses make Macros so only one line needs to be changed to add a file that needs to be compiled and linked. Depend lines may be added or changed to get the dependencies correct.
– Add a “clean” target that removes any file created by the makefile. This includes ush.o and ush.
Also, it your Makefile must use the built-in suffix rules for compiling C programs as is the case with the simple makefile above. (Notice in the given “makefile” above, it does not specify how to make “ush.o”, it just knows how to do it. This is due to the built-in suffix rules.)

As a final note, make sure you do not add ush.o or any other .o file or ush to you git repository. Use a gitignore file to help you ignore files that should not be in your repository.
• (80 points) Add environment variable processing. Do this by:

– Create a new C source file called “expand.c” and add a new function expand() in this file. The prototype of expand() must be:

```c
int expand (char *orig, char *new, int newsize);
```

The parameter “orig” is the input. The parameter “new” should be a pointer to a fixed size array similar to buffer in main(). The parameter newsize is the number of characters in the new array. The parameter newsize is very important. This is an “input parameter”. It tells your expand() function the maximum numbers of characters that may safely be put into the parameter new. Your code must make sure that expand() does not overflow the new array by using the newsize parameter. The return int should be a value that means “expand was successful” or “expand had an error and processline should stop processing this line”). “expand()” should not change the original line. (Temporary changes are allowed as long as the array remains unchanged at the point of return from expand().) This function processes the original line as explained below and produces a new line in the array pointed to by the “new” parameter. The new line must be correctly terminated with the “end-of-string” character.

– Create a new file called “defn.h” that should have the prototype of expand in the file. Then both ush.c and expand.c should include the .h file so they both get the same prototype. Update your Makefile so it correctly compiles ush.c and expand.c. You should also have a depends line to show that ush.c and expand.c both depend on defn.h because they both include it.

Your expand.c file may have helper functions in it, but only the expand() function should have its prototype in defn.h. Also, remember to have a “header comment” in expand.c that describes the content of the file and who wrote the code.

– In the function processline(), you will add a call to “expand()”. This call to expand must be called before arg_parse() because the new line is what must be passed to arg_parse. Notice, as a result of this, the original line passed to processline() is not modified. This is different than assignment 1 in which “line” from main() was modified. Also, you should just declare an array in processline() for the new line. Do not malloc() space for this new line.

– expand() is to process the original line exactly once, starting with the first character (orig[0]) and looping to the last character. It should do this loop exactly once and do all the required processing during this one loop. When expand finds text that needs replacing, it replaces the text with the required new text and copies the new text to the new string. It then must continue processing the original text at the point after the replaced text. Do not expand the new text. The basic processing done by expand() is to copy each character, one at a time, to the new string unless it detects that the current character is part of text
that needs to be replaced. In that case, expand() adds the new text (expansion text) to the new string as specified by the replacement. Be careful to not write code that is $O(n^2)$. Also, remember that strlen(3) is an $O(n)$ operation. You should never need to use strlen(3) on the new string you are constructing.

- In expand(), replace ${NAME}$ in the original string with the value of the environment variable of the same name. (Yes, the braces are part of the syntax. ${NAME}$ is not processed as an environment variable.) This value of the environment variable is placed in the new string. If the name does not exist in the environment then no characters are copied to the new string. The ${NAME}$ characters are also NOT copied to the new string. This is done for all ${..}$s found in the original string. In ALL cases the ${NAME}$ will not appear in the new string. If you find a ${ and do not find a closing }, print an error message and stop processing the line. Read about the library function getenv(3).

- (5 of the 80 points) Prepare for other special variable processing in expand(). Write your code to easily add other kinds of expansion. To show that it is successful, add the following expansion.

$$ is replaced by the base 10 ASCII equivalent of the shell’s pid.

- (85 points) Start a new file for built-in commands, “builtin.c”, and implement built-ins as follows:

  - Build a framework for executing built-in commands. These commands are done directly by the shell and are not forked to another process to do. Your processline() function should call arg_parse() and then call a routine to check to see if the command is a built-in command. This function may execute the built-in if it is a built-in before returning to processline(). You may also have two functions, one to check if the command is a built-in and one to execute it. Add the prototype definitions of the one or two functions to defn.h. These functions should accept both the argument vector and the number of arguments since arg_parse() is giving processline() that information. After processline() determines it is not a built-in command, that is the time to call fork(). If you execute a built-in command, no fork() is called. Remember, your “shell” process must free the argument list after it is done using it, regardless of whether it uses fork() or not. Note: in later assignments you will be required to add more built-in commands. A good “framework” will provide a very easy way to add new built-in commands. While a series of “if (strcmp(......)...) ... else if (strcmp(......)...) ...” will work, try to think of a way to to use strcmp only once (in a loop) to determine which built-in command needs to be executed. (Function pointers are really nice to use here with having each built-in command in its own function. If you have never worked with function pointers, look at the files named “funcptr.c” and “funcptr1.c” in the directory /home/phil/public/csci347.)

  - All the identification and execution of built-in commands must be implemented in code in your builtin.c file. You should have at most two functions that processline() calls directly
to implement your built-in commands. If you have two functions, one should answer the question “is the command described by this argv a built-in function” and the other one should do the built-in command. A single function would combine the functionality of the two into a single function. Add the prototype for your built-in commands function(s) to your “defn.h”.

Note: if you are calling strcmp() in processline, you are not implementing this as requested and will lose points. Also, if you put definitions in ush.c to help you with built-in commands, you are also not doing as requested. All code implementing built-in commands must be in builtin.c. This includes global variables for built-in commands. These global variables must be static. No prototypes of “helper functions” in builtin.c should be in defn.h, just the routines that are called from processline().

- Implement exactly the following four built-in commands. (The brackets ([...]) in the descriptions do not appear in the real command, they just show that that part is optional.) Remember to check for errors and if an error is found, print an error message and then stop processing and return to processline().
  * exit [value] - exit the shell with the value. If value is not given, exit with value 0. This built-in command should call the exit(3) library call. Hint: man 3 atoi
  * envset NAME value that sets the environment variable of the same name to the given value.
  * envunset NAME that removes the variable NAME from the current environment.
  * cd [directory] Use chdir(2) to change the working directory of the shell. If the directory is not given, use the environment variable HOME. Any errors should report the error and then do nothing.

- (15 points) The final 15 points are available in following the style guidelines and how you turn in your assignment. Here are some final notes for this assignment and the specification of how to turn in your assignment.

  - None of your source files should #include a .c file and no executable code may appear in your .h file(s).

  It is possible to compile all three files on one command like:
  ```
  cc -g -Wall -o ush ush.c builtin.c expand.c
  ```

  I am requiring that your makefile compiles your sources using several commands like:
  ```
  cc -g -c -Wall ush.c
  cc -g -c -Wall builtin.c
  ```
cc -g -c -Wall expand.c
cc -g -o ush ush.o builtin.o expand.o

- **What to turn in:** You must turn in your sources as a .pdf and turn in a *sample run* where *you* tested your shell. This test must not be running my grading script on your shell. You will get graded on the quality of the tests you turn in. Also, you must turn in a script of you running the grading script. Make sure it is easy to determine which are your tests and what is the grading script. Just to repeat from assignment 1, there are a number of programs on the departmental Linux labs that can help you. “a2ps” turns text into a postscript file, “ps2pdf” turns postscript files into pdf files. “pdfunite” and “pdfjoin” allows you to create one pdf file from several. “script” captures your shell session. “soffice” can create pdf files and can read text files.

- My reference executable is available for you to run. On the lab machines, you may run the program `/home/phil/.bin/ush`. If you have a question like “What should the shell do if I type ‘.....’”, ask my shell instead of asking me. Of course, if you don’t understand what it does, ask me.

- When you ask questions via e-mail or come to my office, please commit and push your current work. This should be on branch “main”. With your current work pushed to gitlab, it is much easier for me to see your code and help you.

- I will make my test script available sometime on or after Wednesday April 17, in the directory `/home/phil/public/csci347/testa2` on lab machines. To use use the script, cd to the `/home/phil/public/csci347/testa2` directory and give the command `./try` when there. If you have not set up a ssh key with gitlab, give the command `./try -H` in the directory. If you don’t know if you did set up a ssh key with gitlab, assume you didn’t and use the second form. It will create a directory `/home/username/` where it will test your work. (username is your user name.) It is possible that not all tests done by my real grading script are done by the public one. The public one does most of them. If your program passes all my tests, the script says

```
  Script output same
  Exit values correct
```

- When you have **completed this assignment and are ready to turn it in**, at that point, create branch ‘a2’ and push it to origin. (If you forget how to push a new branch, the assignment 1 write up has the instructions.)

- Once you have created the ‘a2’ branch, turn in your .pdf file on canvas. You should have no commits on your a2 branch. As a reminder, you should submit a single .pdf file and that .pdf file should have a cover sheet, your source code and your two test runs.