For this assignment, you will continue work on the micro-shell. Start with the code base you have for the last assignment, a3. Continue using branch master for your work.

- **(25 points)** Expand `?` by replacing it with the base 10 ascii derived in the following way:
  - If the command was a built-in command, use 0 for a successful built-in and 1 for an unsuccessful built-in command.
  - If the command called `exit(2)`, use the value passed passed to that function.
  - If the command was killed by a signal, use the number 128 plus the signal number.

An “easy” implementation has the `wait` code and built-in commands set a global variable to the correct value to be used by `expand()`.

- **(20 points)** Upgrade your parent code in `processline()` to report when a child has been terminated by a signal other that SIGINT. It should also report a “core dump” if one happened in parenthesis, similar to bash. Do not report for commands started in command expansion ($()) and report only for the last element of a pipeline (see below.)

- **(100 points)** Add command output expansion to `expand()`. In expanding a command line, if you find a `$(`, find the matching `) `and everything between the `$( `and the matching `)` is considered a shell command and executed. Notice, count every `(` in your matching, not just the ones preceded by `$`. The characters `$(cmd ...)` are replaced with the standard output of the command “cmd ...” after you replace each newline (`\n`) character in the command output with a space. If the final character in the command output is a newline character, it should not be replaced with a space, it should be deleted. Do not do expansion on the output of the command. Notice, all standard processing for the command inside the $( must be done, including all expansion and argument processing. The easiest way to process the $( command is to find the terminating `)` and replace it with the null character (\0), call processline() on the command text and after the call, put back the ) into the command string. (This implies that the expansion function will be indirectly called recursively.) You must allow for at least 200000 characters in the final command expansion. (This may be a compile constant.) (Warning: Be careful when allocating a 200000 character local variable in functions that may be called recursively. If you use more than one of these
big arrays in processline and expand, you may have your program crash on you when using recursion. On Linux, you have 8 Megabytes for your stack size so that gives you the maximum of about 40 of these 200000 character arrays.

As an example, consider the following:

```
% pwd
/home/phil
% envset PWD $(pwd)
% echo ${PWD}
/home/phil
% echo The current working directory is $(pwd).
The current working directory is /home/phil.
% envset VAR 1
% envset VAR $(expr ${VAR} + 1)
% echo ${VAR}
2
```

Also note, some shells also recognize this expansion with the syntax `cmd ...`. You are to implement only the $() version. Also, notice that the $(...) version allows for “embedded commands”. For example,

```
% echo $(ls -ld $(dirname $(pwd)))
```

Finally, you must use pipes to implement $() and you must use only one pipe for each $(). A correct implementation should have at most two processes running at any time in processing $() and this includes nested $( $() ) forms. Builtin commands in $() must be run by the shell without forking and you do not need to worry about deadlock with builtin commands for this assignment. $() commands also sets the value to use for $? expansion. And if you use popen(3) to implement $(), you will lose 90 points of the possible 100 points.

- (10 points) Add signal processing so that your shell is not terminated by SIGINT. Warning! Some system calls are interrupted by a signal and return with errno set to EINTR. You may need to deal with these depending on how you set up things. Also, your signal handler should not just siglongjmp(3) back to main. The best way to do this is to have your signal handler send SIGINT to any process on which your shell is wait(2)ing and set a global variable that says a SIGINT happened. This is important for command expansion so that if a SIGINT happens while processing the command expansion, expand can recognize the signal and stop processing.
• (80 points) Add pipelines to your shell. For example:

```
% finger | tail -n +2 | cut -c1-8 | uniq | sort
```

should print out a sorted list of user names of all logged in users. The return value for
the shell variable $? is the exit value of the last command in the pipeline. Parent code
report signals only for the last command in the pipeline. Pipes do not pipe standard
error, it only connects the standard output of the command on the left of the | to the
standard input of the command on the right.

*Note:* Make sure your pipelines do not generate zombies. You must properly “wait()”
(or wait3, wait4 or waitpid) on all children of a pipeline.

Notes:

• 15 points of this assignment are for the turn in, style and so forth.

• Again, I will grade your assignments by running a script that checks out your sources,
runs make and then runs a series of scripts.

• As usual, the student version of the grading scripts will be available in the directories
/home/phil/public/csci347/testa4. If it is not readable two days before the assignment
is due, send me e-mail to remind me to make them readable.

• Please turn in your sources on paper. Don’t forget scripts with your own tests of your
shell and a run of the grading script. You may lose points if you don’t show enough
tests of your devising.

• It is best to implement $() before pipelines.

• Notice where common tasks are done and make them functions!

• Use recursion where it helps you!

• Full recursion for pipelines can be difficult, but it can work. I recommend you use
iteration for pipelines even though it still needs a recursive call to processline().

• Start early, work often. Don’t wait to start this assignment.

• Pipelines can appear in the $() commands!

• If you have a question of what does a particular command do, try it using my shell. It
is found at /home/phil/.bin/ush on the lab machines.