

## 1 Overview

In this project, you will write the guts of a shell that will support boolean operations, pipes, and file redirection.

## 2 Procedure

### 2.1 Obtain the project files

We have supplied several files for your use in this project:

**command.h** defines the tree structure that the parser will produce from a command line, and the conjunctions that join commands into a line.

**d8sh.c** is the main shell loop. You need not modify this file, but it may be useful to inspect for generating test cases.

**executor.c** is where you must implement the function `int execute(struct tree *t)`, which will take a tree from the parser and execute the commands in the tree. The return value of `execute` may be used as you like, e.g., for exit status or for a process id, if you choose to call `execute` recursively.

**executor.h** is the declaration of `execute`, used by the parser.

**lexer.c** is the lexer, which splits a command line into tokens, generated by flex.

**lexer.h** declares lexer related functions referenced by `d8sh`.

**parser.tab.c** is the parser, which assembles tokens into a parsed tree, generated by bison.

**parser.tab.h** is a part of the parser, generated by bison, referenced by `lexer.c`.

**run-all-tests.csh** checks for Makefile and `d8sh`, then compares the output of your `d8sh` using the command lines in the testing subdirectory.

These files are contained in `~/216public/project6`. Your code for this project should be contained in your `~/216/project6` subdirectory.

### 2.2 Create a Makefile

Create a Makefile that we will use to build your shell. Section 3.1 lists the targets you are required to implement, as well as other requirements for your Makefile.

### 2.3 Implement and test the shell

You must implement your shell program by supplying the necessary code in `executor.c`. As you implement various features of the shell, you can test them by either interactively running `d8sh` while typing in commands, or you can create text files with one command per line, and use redirection to feed them as input to your shell.

The video <https://tinyurl.com/yadjttla> can help you understand what you need to implement for this project. Start watching the video at time 19:13.

## 3 Specifications

### 3.1 Makefile

Your Makefile should be set up so that all programs are built using separate compilation (i.e., source files are turned into object files, which are then linked together in a separate step). All object files should be built using

the class flags.

You must have the following targets in your Makefile:

1. `all`: make all executables
2. `clean`: delete all object files and executables
3. `lexer.o`, `parser.tab.o`, `executor.o`, and `d8sh.o`, the object files for the parsing code and shell code
4. `d8sh`: the executable created by linking the parsing object file with the shell object files and linked with the readline library using `-lreadline`.

We will use the Makefile you provide to build the public test executables on the submit server; if there is no Makefile, your shell program will not be built, and you will not receive credit for **any** tests.

You may have other targets, for example `"test"`.

## 3.2 The Shell

The shell you will implement will include some, but not all features, of the shell you use on grace. The features your shell supports will include pipes, input and output redirection, and the `"&&"` operator.

The parser may permit a few more operations that you need not support. In particular, the `"||"` and `;"` operators exist in the format, but need not be supported by your code.

To invoke the parser, `d8sh` uses `readline()`, then `yy_scan_string(buffer)`, followed by `yyparse()`. (Typically, a bison parser would load a file such as your C source. Here, we are using this system to operate on a string.) When the parser constructs a complete tree from a command line, it invokes your `execute()` function.

You are given the main shell program in `d8sh.c`: your task is to implement the `execute()` function called by `main()`. Once the full `d8sh` program is linked together from the four object files, `d8sh` should function as a normal shell, with some limitations as described below.

The syntax for the features we want you to implement is very similar to the shell syntax you should be familiar with from your experience working with the Unix environment. These are the specific features your shell must provide:

1. **File redirection**: by using the `<` and `>` tokens, a user should be able to redirect standard input and output in the same manner as is done in the `tcsh` and `bash` shells.

If a file is created by output redirection, the file should be created using permissions `0664`.

In the case of a multi-program pipeline, file input redirection may be applied to the first program (before the first pipe character), and file output redirection may be applied to the final program. A shell must print `"Ambiguous input redirect."` or `"Ambiguous output redirect."` if a user tries to provide a redirected file and a pipe at the same time. For example, the following is illegal `"echo hello > x | cat"`.

2. **Piping**: if programs are separated by pipe characters (`"|"`) on a command line, then the standard output of the program to the left of the pipe character is to be connected to the standard input of the program to the right of the pipe character, creating a pipeline.

When running a pipeline, the shell must start all of the processes, but not return to print another prompt until all processes have exited. You will need to determine how to fork all the processes piped together and then wait for them all to finish.

3. **Subshells**: when expressions are surrounded by parentheses, the command is executed in its own shell, forked from the parent. This ensures that environment changes (such as changing the current directory) are contained to the subshell.

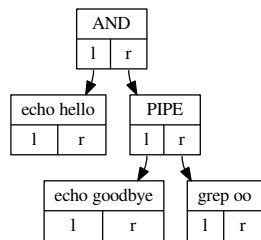
As you can see from the code in `d8sh.c`, the shell prompts the user for a command, parses the command, and then attempts to execute the command. To execute the command, you must use the `struct tree *` parameter to see which options are set, and perform the steps necessary to execute the command as requested.

Should you encounter any errors in executing the command, your shell **must not terminate** – children of the shell may terminate, but the code you write in `execute()` must not cause the parent (shell) process to terminate. A user should not cause the shell to die because he/she, for example, mistyped the name of a program to execute.

If your shell cannot exec any process, it should print, “Failed to execute %s” with the name of the program that failed. Other failed system calls should print using `perror`, e.g., `perror("fork")`. The shell is already configured for parse errors to print as “Parse error: %s” via the `yyerror()` function.

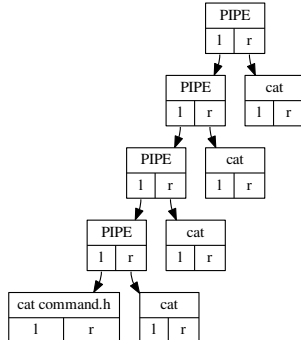
## 4 Example Trees

You may want to print out the contents of each parse tree you work on. Here are some examples.



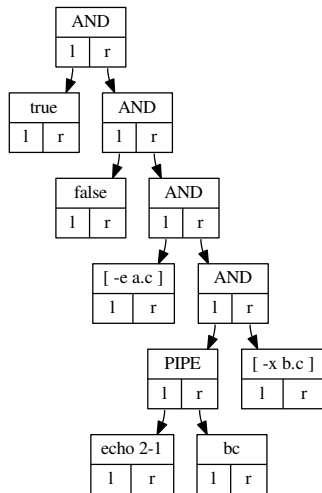
```
echo hello && echo goodbye | grep oo
```

Note that the pipe operation has a higher precedence than `and`. As a result, “hello” is printed to stdout and does not pass through `grep`. The `grep` command prints all lines that include a given string, so will print “goodbye” because it includes “oo”.



```
cat command.h | cat | cat | cat | cat
```

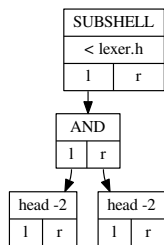
The “`cat`” tool can either read input from arguments or from stdin, then pass it to stdout. Using pipe, fork, `dup2`, and `exec` to build pipelines in tree form can be subtle. The contents of `command.h` should appear on the output. (This is a better example than it is a test; tests may use `tac` instead of `cat` to reverse the file.)



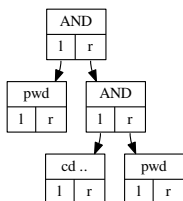
```
true && false && [ -e a.c ] && echo 2-1 | bc && [ -x b.c ]
```

Again, pipe has a higher precedence. Execution should stop at the first false. `true` runs the `/usr/bin/true` program, whose only purpose is to `exit(0)`, a successful result. (See: “man true” and “man false” for more information. There are a few trivial programs like this, for example, “yes” and “seq”. You must use the `wait()` or `waitpid()` functions to determine whether the program exited successfully).

```
(head -2 && head -2) < lexer.h
```

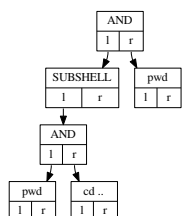


Note how the shell prints the first four lines of `lexer.h` if given this command. Beware, however, the first `head` command might use buffered I/O and thereby slurp more data than just the first two lines, leaving the second `head` command to print two lines from the middle of a longer file. (It does work on `grace` and `submit`.) More reliable would be to redirect output instead of input.



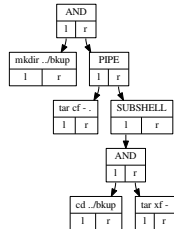
```
pwd && cd .. && pwd
```

This will output the current directory, change to its parent, and output the parent. For the next command, the current directory will be the parent.



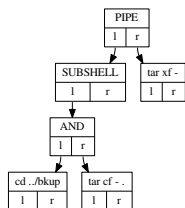
```
(pwd && cd ..) && pwd
```

This will output the current directory and change to its parent in a subshell, then output the current directory again (unchanged). For the next command, the current directory will unchanged.



```
mkdir ../bkup && tar cf - . | (cd ../bkup && tar xf -)
```

This is an example of using `cd` in a subshell in practice to backup a directory. The `cd` affects only the second tar (extract rather than create). This creates a directory, `../otherdir`, then copies the contents of the current directory into it using `tar`. The current working directory after this command is unchanged (not `bkup`). From a subshell, environment (working directory) changes do not escape.



```
(cd ../bkup && tar cf - .) | tar xf -
```

If the previous example created a backup, this one restores it. The `cd` affects only the first tar, leaving the current working directory for the next command where it was. The command will fail if “`bkup`” is not present.

## 5 Important Points and Hints

1. Start by handling nodes with conjunction NONE, i.e., no pipes, no ands. Add redirection, pipes, and the and operation incrementally.
2. `execute()` must implement internally the commands “cd” and “exit”. The “cd” command should change to the user’s home directory (`getenv("HOME")`) if given no arguments.
3. Be sure to wait for **all** child processes of your shell to complete; failing to do so will cause a collection of zombie processes to accumulate and tie up Grace system resources.
4. Be careful that any fork loops you write terminate at some point, before a Grace system administrator has to kill them.
5. You may wish to print the contents of the tree before execution. Each node is either an internal node of the tree, representing a PIPE or an AND, or a leaf node, representing a command with `argv` set. Any node, including internal nodes may have input and output redirection. Your shell should mimic real shells for the following commands:

```
(head && head) < lexer.h
(head < lexer.h && head < lexer.h)
(cat command.h | cat - command.h)
(cat < command.h | cat - command.h)
```

6. You are permitted to translate the tree structure into whatever representation you prefer. (This is not encouraged.)
7. You need not free the data or nodes in the parse tree. Yes, this will leak memory.
8. Closing the correct file descriptors for a pipe at the correct time can be tricky. All the file descriptors for writing must be closed before a reader will see the end of file and exit. If the shell hangs, there’s probably a process waiting for more input that will never come.
9. Don’t get excited and modify the shell prompt to make it look more shell like; that may have bad consequences on the submit server.
10. Please consider using git locally to track your changes; for example, supporting AND may break your design for PIPE, and it may be useful to go back in time to find your working solution.
11. Make sure your `executor.c` file has your student id (e.g., 111XXX...) NOT your directory id. There are tests that check for the presence of this id.
12. Make sure you code compiles without warnings, otherwise you will lose credit. Make sure you comment out the `print_tree` function provided with `executor.c` once you have no longer use for it. There are tests that check for warnings.
13. You need to add to `executor.c` any `.h` files that allow you to fork, `exec*`, etc.
14. The message “Failed to execute %s” ... should be sent to standard error.
15. Use standard output for ambiguous output redirect and ambiguous input redirect messages.
16. If `chdir` fails, use `perror` to print an error message. For example, `perror(location)` where `location` is the directory.
17. If both ambiguous input and output redirect take place, only one message will be printed ( “Ambiguous output redirect.” )

## 6 Grading Criteria

Your project grade will be determined by the following formula:

Results of public tests	40%
Results of secret tests	60%

The public tests will be made available shortly after the project is released. You can find out your results on these tests by checking the submit server a few minutes after submitting your project. Secret tests, and their results, will not be released until after the project's late deadline has passed.

Public tests for this project consist of input files to be read by your shell via standard input, and checked against the expected output, for example like this:

```
./d8sh < public00.in | diff - public00.output
```

We've put the public tests in the testing subdirectory, and some of these tests use extra input files from the current directory. The `run-all-tests.csh` script temporarily links those files to the current directory. Consider inspecting that script for more examples of booleans in shell commands.

## 7 Submission

### 7.1 Deliverables

The only files we will grade are (a) your `Makefile`; and (b) `executor.c`, which contains your shell implementation. We will use our versions of all other files to build tests, so do not make any changes to other files.

### 7.2 Procedure

To limit the number of processes your shell implementation can create when you run it on the Grace machines, ensure that the line `limit maxproc 20` is present in the `.cshrc.mine` file in your home directory. This is part of the 216 setup script. Run "limit" to check that it is set.<sup>1</sup>

As for previous projects, executing "submit" in your project directory will submit your project. We encourage you to run public tests on Grace rather than submitting to the submit server and waiting for your submission to be evaluated; it is much faster for you to see your results if you run the tests yourself, and the submit server works much more quickly if the class makes fewer submissions.

## 8 Other Notes

### 8.1 Academic Integrity

As noted in the syllabus, any evidence of cheating will be referred to the Student Honor Council and may result in a grade of XF in this course. Submissions will be checked with an automated source code comparison tool to look for evidence of copying, collaborative work, and use of prohibited online materials.

### 8.2 Deadlines

Submission deadlines are strictly enforced by the submit server, and we will not extend them for events such as network outages. Extensions may be given on a case-by-case basis, but will likely only be granted for medical or other unforeseeable cases. Therefore, you should start work on this project soon, as last-minute technical problems are not an excuse for missing either the on-time or late deadline.

---

<sup>1</sup>If you use `bash` instead of `tcsh`, the command is `ulimit -u 20`.