In this assignment, you will explore many important process concepts by implementing a POSIX-like shell program. Your program will:

1. Parse command-line input into commands to be executed
2. Execute a variety of external commands (programs) as separate processes
3. Implement a variety of shell built-in commands within the shell itself
4. Perform a variety of i/o redirection on behalf of commands to be executed
5. Assign, evaluate, and export to the environment, shell variables
6. Implement signal handling appropriate for a shell and executed commands
7. Manage processes and pipelines of processes using job control concepts

1 Learning Outcomes

- Describe the Unix process API (Module 3, MLO 2)
- Write programs using the Unix process API (Module 3, MLO 3)
- Explain the concept of signals and their uses (Module 3, MLO 2)
- Write programs using the Unix API for signal handling (Module 3, MLO 3)
- Explain I/O redirection and write programs that can employ I/O redirection (Module 3, MLO 4)

2 Overview

You will write a program, named BigShell, that implements a large subset of the POSIX Shell's functionality.

The high-level functionalities you will be tasked with implementing are:

- The following built-in shell utilities:
  - **`cd`** – Change the working directory
  - **`exit`** – Cause the shell to exit
  - **`unset`** – unset values of variables
- Signal handling behavior of the shell, and commands
- Command execution
  - Expansion of:
    - Command words
    - Assignment values
    - Redirection filename operands
  - I/O Redirection of the following operators:

    `* >` `* <` `* <>` `* >>` `* >|` `* >&` `* <&`

  - Variable assignment
  - Simple commands:

        * Foreground commands – `;`
        * Background commands – `&`
    - Shell pipelines – `|`
    - Job control

3 Reference Documents

The overall assignment is loosely based on the shell command language specification in POSIX.1-2008, with several features removed or simplified.

Builtins

- *cd(1)*
    - *chdir(3)*
- *exit(1)*
    - *exit(3)*
    - *strtol(3)*
- *unset(1)*
    - *getenv(3)*/*setenv(3)*

Signal Handling

- *sigaction(3)*
- *kill(2)*
- *signal.h(0)*
- *signal(7)*

I/O Redirection

- *dup(2)*
- *fcntl(3)*
- *open(2)*

Foreground/Background Commands

- *wait(2)*

Pipelines

- *pipe(2)*

Job Control

- *isatty(3)*
- *setpgid(3)*/*setpgrp(3)*
- *getpgid(3)*/*getpgrp(3)*
- *tcsetpgrp(3)*/*tcgetpgrp(3)*

General Purpose

- *ctype.h(0)*
- *errno.h(0)*

4 Command Language

4.1 BigShell Command Language

This chapter contains the definition of the BigShell Command Language.

4.1.1 BigShell Introduction

BigShell is a command language interpreter. This chapter describes the syntax of that command language as it is used by the BigShell utility.

BigShell operates according to the following general overview of operations. The specific details are included in the cited sections of this chapter.

1. BigShell reads its input from standard input.
2. BigShell breaks the input into tokens: words and operators; see *Token Recognition*.
3. BigShell parses the input into commands (see *Shell Commands*).
4. BigShell performs various expansions (separately) on different parts of each command, resulting in a list of pathnames and words to be treated as a command and arguments (see *Word Expansions*).
5. BigShell performs redirection (see *Redirection*) and removes redirection operators and their operands from the parameter list.
6. BigShell executes a built-in (see *Special Built-In Utilities*), or executable file, giving the names of the arguments as positional parameters numbered 1 to *n*, and the name of the command as the positional parameter numbered 0 (see *Command Search And Execution*).
7. BigShell optionally waits for the command to complete and collects the exit status (see *Exit Status*).

4.1.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to BigShell. Quoting can be used to preserve the literal meaning of the special characters in the next paragraph, and prevent parameter expansion.

The application shall quote the following characters if they are to represent themselves:

```
|  &  ;  <  >  $  \  "  '  <space> <tab>
```

The `<newline>` character cannot be quoted.

The various quoting mechanisms are the escape character, single-quotes, and double-quotes.

Escape Character (Backslash)

A <backslash> that is not quoted shall preserve the literal value of the following character, with the exception of a <newline>. If a <newline> follows the <backslash>, BigShell shall interpret this as line continuation. The <backslash> and <newline> shall be removed before splitting the input into tokens. Since the escaped <newline> is removed entirely from the input and is not replaced by any white space, it cannot serve as a token separator.

Single-Quotes

Enclosing characters in single-quotes (`''`) shall preserve the literal value of each character within the single-quotes. A single-quote cannot occur within single-quotes.

Double-Quotes

Enclosing characters in double-quotes (`""`) shall preserve the literal value of all characters within the double-quotes, with the exception of the characters <dollar-sign>, and <backslash>, as follows:

$
: The <dollar-sign> shall retain its special meaning introducing parameter expansion (see *Parameter Expansion*).

  Tee string of characters from an enclosed `"${"` to the matching `'}'` are parsed literally.

\`
: The <backslash> shall retain its special meaning as an escape character (see *Escape Character (Backslash)*) only when followed by one of the following characters when considered special:

  ```
  $  "  \  <newline>
  ```

The application shall ensure that a double-quote is preceded by a <backslash> to be included within double-quotes.

4.1.3 Token Recognition

BigShell shall read its input in terms of lines from stdandard input, which may be a terminal, in the case of an interactive shell.

The input lines can be of unlimited length. These lines shall be parsed using two major modes: ordinary token recognition and processing of here-documents.

BigShell shall break its input into tokens by applying the first applicable rule below to the next character in its input. The token shall be from the current position in the input until a token is delimited according to one of the rules below; the

characters forming the token are exactly those in the input, including any quoting characters. If it is indicated that a token is delimited, and no characters have been included in a token, processing shall continue until an actual token is delimited.

1. If the end of input is recognized, the current token shall be delimited. If there is no current token, the end-of-input indicator shall be returned as the token.
2. If the previous character was used as part of an operator and the current character is not quoted and can be used with the current characters to form an operator, it shall be used as part of that (operator) token.
3. If the previous character was used as part of an operator and the current character cannot be used with the current characters to form an operator, the operator containing the previous character shall be delimited.
4. If the current character is <backslash>, single-quote, or double-quote and it is not quoted, it shall affect quoting for subsequent characters up to the end of the quoted text. The rules for quoting are as described in *Quoting*. During token recognition no substitutions shall be actually performed, and the result token shall contain exactly the characters that appear in the input (except for <newline> joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the <quotation-mark> and the end of the quoted text. The token shall not be delimited by the end of the quoted field.
5. If the current character is an unquoted `'$'`, BigShell shall identify the start of parameter expansion (*Parameter Expansion*), introduced with the unquoted character sequences: `'$'` or `"${"` . BigShell shall read sufficient input to determine the end of the unit to be expanded (as explained in the cited section). The token shall not be delimited by the end of the substitution.
6. If the current character is not quoted and can be used as the first character of a new operator, the current token (if any) shall be delimited. The current character shall be used as the beginning of the next (operator) token.
7. If the current character is an unquoted <newline>, the current token shall be delimited.
8. If the current character is an unquoted <blank>, any token containing the previous character is delimited and the current character shall be discarded.
9. If the previous character was part of a word, the current character shall be appended to that word.
10. If the current character is a `'#'` , it and all subsequent characters up to, but excluding, the next <newline> shall be discarded as a comment. The <newline> that ends the line is not considered part of the comment.
11. The current character is used as the start of a new word.

4.1.4 Parameters and Variables

A parameter can be denoted by a name, or one of the special characters listed in *Special Parameters*. A variable is a parameter denoted by a name.

A parameter is set if it has an assigned value (null is a valid value). Once a variable is set, it can only be unset by using the *unset* special built-in command.

Special Parameters

Listed below are the special parameters and the values to which they shall expand. Only the values of the special parameters are listed.

?

Expands to the decimal exit status of the most recent pipeline (see *Pipelines*).

$

Expands to the decimal process ID of the invoked shell.

!

Expands to the decimal process ID of the most recent background command (see *Asynchronous Lists (Background Commands)*). For a pipeline, the process ID of `$?` is that of the previously executed command in the pipeline.

Shell Variables

Variables shall be initialized from the environment. If a variable is initialized from the environment, it shall be marked for export immediately; see the *export* special built-in.

The following variables shall affect the execution of BigShell:

HOME

The pathname of the user's home directory. The contents of *HOME* are used in tilde expansion (see *Tilde Expansion*).

PATH

A string formatted as described in XBD Environment Variables, used to effect command interpretation; see *Command Search and Execution*.

PS1

Each time an interactive shell is ready to read a command, the value of this variable shall be written to standard error. The default value shall be `"$"` for regular users, and `"#"` for the root user.

PS2

Each time the user enters a <newline> prior to completing a command line in an interactive shell, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value is `">"` .

PWD

Set by the cd utility.

4.1.5 Word Expansions

This section describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained in the following sections.

The order of word expansion shall be as follows:

1. Tilde expansion (see *Tilde Expansion*) and parameter expansion (see *Parameter Expansion*) shall be performed, beginning to end. See item 5 in *Token Recognition* .
2. Quote removal (see *Quote Removal*) shall always be performed last.

The `'$'` character is used to introduce parameter expansion. If an unquoted `'$'` is followed by a character that is either not the name of one of the special parameters (see *Special Parameters*), a valid first character of a variable name, or a <left-curly-bracket> (`'{'`), the result is unspecified.

Tilde Expansion

A "tilde-prefix" consists of an unquoted <tilde> character at the beginning of a word, followed by all of the characters preceding the first unquoted <slash> in the word, or all the characters in the word if there is no <slash>. The characters in the tilde-prefix following the <tilde> are treated as a possible login name from the user database.

If the login name is null (that is, the tilde-prefix contains only the tilde), the tilde-prefix is replaced by the value of the variable *HOME*, if it is set. Otherwise, the result is unspecified.

If the login name is specified, the tilde-prefix shall be replaced by a pathname of the initial working directory (home directory) associated with the login name, obtained using the getpwnam() function as defined in the System Interfaces volume of POSIX.1-2008. If the system does not recognize the login name, the results are undefined.

Parameter Expansion

The format for parameter expansion is as follows:

```
${expression}
```

where *expression* consists of all characters until the matching `'}'` .

The simplest form for parameter expansion is:

```
${parameter}
```

The value, if any, of *parameter* shall be substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for when *parameter* is followed by a character that could be interpreted as part of the name.

If the parameter name or symbol is not enclosed in braces, the expansion shall use the longest valid name (see XBD Name), whether or not the symbol represented by that name exists.

Quote Removal

The quote characters (<backslash>, single-quote, and double-quote) that were present in the original word shall be removed unless they have themselves been quoted.

4.1.6 Redirection

Redirection is used to open and close files for the current shell execution environment (see *Shell Execution Environment*) or for any command. Redirection operators can be used with numbers representing file descriptors (see XBD File Descriptor) as described below.

The overall format used for redirection is:

```
[n]redir-op word
```

The number *n* is an optional decimal number designating the file descriptor number; the application shall ensure it is delimited from any preceding text and immediately precede the redirection operator *redir-op*. If *n* is quoted, the number shall not be recognized as part of the redirection expression. For example:

```
echo \2>a
```

writes the character 2 into file **a**. If any part of *redir-op* is quoted, no redirection expression is recognized. For example:

```
echo 2\>a
```

writes the characters 2>*a* to standard output. The optional number, redirection operator, and *word* shall not appear in the arguments provided to the command to be executed (if any).

Open files are represented by decimal numbers starting with zero. The largest possible value is implementation-defined. These numbers are called “file descriptors”. The values 0, 1, and 2 have special meaning and conventional uses and are implied by certain redirection operations; they are referred to as *standard input*, *standard output*, and *standard error*,

respectively. Programs usually take their input from standard input, and write output on standard output. Error messages are usually written on standard error. The redirection operators can be preceded by one or more digits (with no intervening <blank> characters allowed) to designate the file descriptor number.

If more than one redirection operator is specified with a command, the order of evaluation is from beginning to end.

A failure to open or create a file shall cause a redirection to fail.

Redirecting Input

Input redirection shall cause the file whose name results from the expansion of *word* to be opened for reading on the designated file descriptor, or standard input if the file descriptor is not specified.

The general format for redirecting input is:

```
[n]<word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard input (file descriptor 0).

Redirecting Output

The two general formats for redirecting output are:

```
[n]>word
[n]>|word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard output (file descriptor 1).

Output redirection using the `'>'` format shall fail if the file named by the expansion of *word* exists. Otherwise, redirection using the `'>'` or `">|"` operators shall cause the file whose name results from the expansion of *word* to be created and opened for output on the designated file descriptor, or standard output if none is specified. If the file does not exist, it shall be created; otherwise, it shall be truncated to be an empty file after being opened.

Appending Redirected Output

Appended output redirection shall cause the file whose name results from the expansion of word to be opened for output on the designated file descriptor. The file is opened as if the open() function as defined in the System Interfaces volume of POSIX.1-2008 was called with the O_APPEND flag. If the file does not exist, it shall be created.

The general format for appending redirected output is as follows:

```
[n]>>word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection refers to standard output (file descriptor 1).

Duplicating a File Descriptor

The redirection operators:

```
[n]<&word
[n]>&word
```

shall duplicate one file descriptor from another, or shall close one.

If *word* evaluates to one or more digits, the file descriptor denoted by *n*, shall be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for input, a redirection error shall result.

If *word* evaluates to `'-'` , file descriptor *n*, or standard input if *n* is not specified, shall be closed.

If *n* is not specified, it shall default to `0` for the `<&` operator, and `1` for the `>&` operator.

If an attempt is made to close a file descriptor that is not open, the behavior is undefined. If *word* evaluates to something else, the behavior is unspecified.

Open File Descriptors for Reading and Writing

The redirection operator:

```
[n]<>word
```

shall cause the file whose name is the expansion of *word* to be opened for both reading and writing on the file descriptor denoted by *n*, or standard input if *n* is not specified. If the file does not exist, it shall be created.

4.1.7 Exit Status

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities is documented in this section. The exit status of the standard utilities is documented in their respective sections.

If a command is not found, or execution fails, the exit status shall be 127.

If a command fails during word expansion or redirection, its exit status shall be greater than zero. It shall not be executed if either operation fails.

If a command exits due to a signal, its exit status shall be 128+*n*, where *n* is the signal number.

4.1.8 Shell Commands

This section describes the basic structure of shell commands. The following command descriptions each describe a format of the command.

A *command* is one of the following:

- Simple command (see *Simple Commands*)
- Pipeline (see *Pipelines*)

Unless otherwise stated, the exit status of a command shall be that of the last simple command executed by the command.

Simple Commands

A “simple command” is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by command words and redirections, terminated by a control operator.

The following expansions, assignments, and redirections shall all be performed from the beginning of the command text to the end:

1. The words of the command, variable assignment values, and redirection filename operands, shall be expanded.
2. Redirections shall be performed as described in *Redirection* .
3. Each variable assignment shall be evaluated to assign the corresponding value to the named parameter.

If no command name results, variable assignments shall affect the current execution environment. Otherwise, the variable assignments shall be exported for the execution environment of the command and shall not affect the current execution environment (except for special built-ins).

If there is a command name, execution shall continue as described in *Command Search and Execution* . Otherwise the command shall complete with a zero exit status

Command Search and Execution

If a simple command results in a command name and an optional list of arguments, the following actions shall be performed:

1. If the command name does not contain any <slash> characters, the first successful step in the following sequence shall occur:
   - If the command name matches the name of a special built-in utility, that special built-in utility shall be invoked.
   - Otherwise, the command shall be searched for using the *PATH* environment variable as described in XBD Environment Variables

   If the search is successful, BigShell executes the utility in a separate utility environment (see *Shell Execution Environment*) with actions equivalent to calling the execve() function as defined in the System Interfaces volume of POSIX.1-2008 with the *path* argument set to the pathname resulting from the search, *arg*0 set to the command name, and the remaining arguments set to the operands, if any.
2. If the command name contains at least one <slash>, BigShell shall execute the utility in a separate utility environment with actions equivalent to calling the execve() function defined in the System Interfaces volume of POSIX.1-2008 with the *path* and *arg*0 arguments set to the command name, and the remaining arguments set to the operands, if any.

If the search is unsuccessful, or execution of the specified utility fails, the command shall fail with a non-zero exit status and write an error message to standard error.

Pipelines

A *pipeline* is a sequence of one or more commands separated by the control operator `'|'` . The standard output of all but the last command shall be connected to the standard input of the next command.

The format for a pipeline is:

```
command1 [ | command2 ...]
```

The standard output of *command1* shall be connected to the standard input of *command2*. The standard input, standard output, or both of a command shall be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command (see *Redirection*).

The last command of a pipeline is either an asynchronous (background) command, or a synchronous (foreground) command.

If the pipeline is terminated by a synchronous command, BigShell shall wait for all commands in the pipeline to complete. If the commands in a pipeline are stopped (as by receipt of a signal), BigShell shall no longer wait for it to complete, and instead treat it as a stopped background command.

Exit Status

The exit status of a pipeline is the exit status of the last command specified in the pipeline.

Asynchronous Lists (Background Commands)

If a command is terminated by the control operator <ampersand> (`'&'`), BigShell shall execute the command asynchronously in a subshell. This means that BigShell shall not wait for the command to finish before executing the next command.

The format for running a command in the background is:

```
command1 & [command2 & ... ]
```

Exit Status

The exit status of an asynchronous list is unspecified.

Sequential Lists (Foreground Commands)

Commands that are separated by a <semicolon> (`';'`) shall be executed sequentially.

The format for executing commands sequentially shall be:

```
command1 [; command2] ...
```

Each command shall be expanded and executed in the order specified.

Exit Status

The exit status of a sequential list shall be the exit status of the last command in the list.

4.1.9 Signals and Error Handling

BigShell shall ignore the signals `SIGTSTP` and `SIGTOU` at all times. It shall ignore the signal `SIGINT` except when reading command input–in which case the `SIGINT` signal will interrupt reading of input, causing BigShell will repeat checking for background processes that have exited, and re-parsing the command input.

The signal actions inherited by a non-builtin command shall be the same as those inherited by BigShell from its parent (they must be restored to their original values before execution of a command).

4.1.10 Job Control

Each pipeline shall be executed with its own unique process group, separate from BigShell or any other pipeline's process group, called a Job. Each job is assigned a separate, unique, Job Id.

If standard input is a terminal, as determined by isatty(), then the following additional behaviors will be implemented:

- When a Job is waited for synchronously, the corresponding process group id shall be made the foreground process group for the terminal, as described in tcsetpgrp()
- At all other times, BigShell shall make itself the foreground process group for the terminal.

Before parsing line(s) of input for commands, BigShell shall (asynchronously) wait for each of the process groups corresponding to any current background jobs. When the last child process of a job exits, BigShell shall print an informative message to stderr (see the skeleton code for the format of these messages).

The special built-in utility *fg* may be used to place a background job in the foreground (and wait on it as a foreground job).

The special built-in utility *bg* may be used to cause a stopped background job to continue running in the background.

4.1.11 Shell Execution Environment

A shell execution environment consists of the following:

- Open files inherited upon invocation of BigShell.
- Working directory as set by **`cd`**
- Shell parameters that are set by variable assignment or from the System Interfaces volume of POSIX.1-2008 environment inherited by BigShell when it begins (see the *export* special built-in)

Utilities other than the special built-ins (see *Special Built-In Utilities*) shall be invoked in a separate environment that consists of the following. The initial value of these objects shall be the same as that for the parent shell, except as noted below.

- Open files inherited on invocation of BigShell, and any additions specified by any redirections to the utility.
- Current working directory
- Variables with the *export* attribute, along with those explicitly exported for the duration of the command, shall be passed to the utility environment variables

The environment of BigShell process shall not be changed by the utility

4.1.12 Special Built-In Utilities

The following "special built-in" utilities shall be supported in BigShell command language. The output of each command, if any, shall be written to standard output, subject to the normal redirection and piping possible with all commands.

The term "built-in" implies that BigShell can execute the utility directly and does not need to search for it.

Variable assignments specified with special built-in utilities remain in effect after the built-in completes; this shall not be the case with a regular built-in or other utility.

Bg

NAME

bg - run jobs in the background

SYNOPSIS

```
bg [job_id]
```

DESCRIPTION

The bg utility shall resume suspended jobs by running them as background jobs. If the job specified by job_id is already running, it shall have no effect.

If job_id is not specified, the lowest numbered job id is used.

Cd

NAME

cd - change directory

SYNOPSIS

```
cd [path]
```

DESCRIPTION

The cd utility shall cause BigShell to change its current working directory to the specified *path*. If no *path* is specified, the BigShell shall change its current working directory to the value of the `$HOME` variable.

Exit

NAME

exit - cause BigShell to exit

SYNOPSIS

```
exit [n]
```

DESCRIPTION

The exit utility shall cause BigShell to exit with the exit status specified by the unsigned decimal integer *n*. If *n* is specified, but its value is not between 0 and 255 inclusively, the exit status is undefined.

If *n* is not specified, the value shall be the exit value of the last (foreground) command executed, or zero if no command was executed.

Export

NAME

export - set the export attribute for variables

SYNOPSIS

```
export [name[=value]]...
```

DESCRIPTION

BigShell shall give the export attribute to the variables corresponding to the specified *name*s, which shall cause them to be in the environment of subsequently executed commands. If the name of a variable is followed by = *word*, then the value of that variable shall be set to *word*.

Fg

NAME

fg - run jobs in the foreground

SYNOPSIS

```
fg [job_id]
```

DESCRIPTION

The fg utility shall move a background job into the foreground. If *job_id* is not specified, fg will use the lowest numbered job that is available.

Jobs

NAME

jobs - display jobs

SYNOPSIS

```
jobs
```

DESCRIPTION

The jobs utility shall display a list of all jobs known to BigShell.

Unset

NAME

unset - unset values and attributes of variables and functions

SYNOPSIS

```
unset [name...]
```

DESCRIPTION

Each variable or function specified by *name* shall be unset.

Unsetting a variable or function that was not previously set shall not be considered an error and does not cause BigShell to abort.

5 Discussion

This program is intended to be a fun project for you to learn fundamental process concepts while also building a featureful program to show off as your portfolio project. Writing a shell is very challenging–the language is difficult to parse, the specification leaves out many important details, and is often ambiguously worded, and the size of a shell's code-base is very large. Accomplishing this on your own is a huge feat!

Neither the provided skeleton code, nor reference implementation, are 100% bug free–in fact, I have found several bugs while writing this documentation after having finished the published implementation. Similarly, your program is not expected to handle every edge case or work perfectly in every situation. There may be aspects of the specification which are ambiguous, and edge cases which produce unexpected results. You are always encouraged to use your best judgement in approaching ambiguities, and if in doubt, ask staff for guidance.

The specification is intended merely as a guideline. Unless explicitly tested by the grading rubric, any other aspects of the program, specified or not, are up to you. The TODOs listed in the skeleton code should serve as a road-map for completion of the assignment. Each TODO should explain in detail what you need to implement in order to get to the same functionality as the reference implementation.

The suggested order of implementation is:

- Built-in Commands
- Non-built-in commands
- Foreground/Background process waiting
- Redirection
- Pipelines
- Variable Assignment
- Signal Handling
- Job Control Functionality

6 Constraints

1. Your program must compile to the `release` build target with the provided makefile.

7 Testing Your Program

A reference implementation is provided with the assignment skeleton code. It has been tested to run in the GitHub Codespaces environment and scores full points on the grading script. It is likely to also run on any recent Linux distribution.

8 Rubric

The example expected test output listed below should serve as a guide. You are encouraged to change variable names, try more complex constructions, and different commands. A rubric item may not be tested *exactly* as shown, but gives a general idea of what to expect–we won't try and surprise you or throw you any curve-balls.

8.1 Built-in Commands [20%]

1. Does the **`exit`** built-in work appropriately? [5%]

```
bigshell$ exit 123
bash$ echo $?
123
```

```
bigshell$ bash -c 'exit 123'    # Set the $? variable to 123
bigshell$ exit
bash$ echo $?
123
```

2. Does the **`cd`** built-in work appropriately? [5%]

```
$ echo $HOME
/home/bennybeaver
$ cd
$ pwd
/home/bennybeaver
$ cd /bin
$ pwd
/bin
```

3. If the *HOME* variable is modified, does the **`cd`** utility respond appropriately? [5%]

```
$ cd
$ pwd
/home/donnyduck
$ HOME=/home/bennybeaver    # Supposing that /home/bennybeaver is a directory that␣
↪exists
$ cd
$ pwd
/home/bennybeaver
```

4. Does the **`unset`** built-in work appropriately? [5%]

```
$ echo $HOME
/home/bennybeaver
$ unset HOME
$ echo $HOME

$ X=12
$ echo $X
12
$ unset X
$ echo $X

$
```

8.2 Parameters and Variables [24%]

1. If a variable is set with no command words, does it persist as an internal (not exported) shell variable? [3%]

```
$ unset X        # Ensure X isn't set yet
$ X=123
$ echo $X        # Show that X is expanded
123
$ printenv X     # Show that X is not exported
$
```

2. If a variable is exported with the **`export`** utility, does a child process that is executed have that variable set in its environment? [3%]

```
$ unset X
$ X=123
$ export X
$ printenv X
123
```

3. If a variable is set as part of a command, does the child process that is executed have that variable set in its environment (and only its environment)? [3%]

```
$ printenv X         # Show that X is unset in the shell
$ X=123 printenv X   # Show that X is added to the environment of the printenv␣
↪command
123
$ printenv X         # Show that X remains unset in the shell
$
```

4. If a foreground command is executed, does the `$?` special parameter get updated appropriately? [3%]

```
$ true
$ echo $?
0
$ false
$ echo $?
1
$ sh -c 'exit 123'
$ echo $?
123
```

5. If a background command is executed, does the `$!` special parameter get updated appropriately? [3%]

```
$ sleep 10 &
$ echo $!
29483
$ pgrep sleep
29483
```

6. If a foreground command is executed, does it **not** modify `$!`? [3%]

```
$ sleep 10 &
$ pgrep sleep
29483
$ echo "hello world"
hello world
```

(continues on next page)

(continued from previous page)

```
$ echo $!
29483
```

7. If the `PATH` variable is changed, does it affect command lookup? [3%]

```
$ echo test
test
$ PATH=
$ echo test
bigshell: No such file or directory
```

8. Does the **cd** built-in update the `PWD` shell variable? [3%]

```
$ cd
$ printenv PWD
/home/bennybeaver
$ cd /bin
$ printenv PWD
/bin
```

8.3 Word Expansions [6%]

1. Are command words expanded properly? [2%]

```
$ X=echo
$ $X test
test
$ Y=hello
$ Z=world
$ $X $Y $Z
hello world
```

```
$ X=testfile
$ DIR=~/$X
$ echo $DIR
/home/bennybeaver/testfile
```

2. Are assignment values expanded properly? [2%]

```
$ X=1234
$ Y=~/$X
$ echo $Y
/home/bennybeaver/1234
```

3. Are redirection filenames expanded properly? [2%]

```
$ X=outfile
$ echo "hello world" >| ~/$X
$ cat /home/bennybeaver/outfile
hello world
```

8.4 Redirection [18%]

> **Note**
>
> Each of the tests shown in this section assume that the file(s) being redirected do not initially exist.

1. > operator creates a new file, and doesn't overwrite existing file? [3%]

```
$ echo test > testfile
$ cat testfile
test
$ echo test2 > testfile
[[some error message]]
$ cat testfile
test
```

2. >| operator creates a new file, and **does** overwrite existing file? [3%]

```
$ echo test >| testfile
$ cat testfile
test
$ echo test2 >| testfile
$ cat testfile
test2
```

3. < operator works appropriately? [3%]

```
$ sh -c 'echo test > testfile'
$ < testfile cat
test
```

4. <> operator works appropriately? [3%]

```
$ sh -c 'echo test > testfile'
$ <>testfile sh -c 'cat; echo test2 >&0'
test
$ cat testfile
test
test2
```

5. >> operator works appropriately? [3%]

```
$ sh -c 'echo hello > testfile'
$ cat testfile
hello
$ >> testfile  echo world
$ cat testfile
hello
world
```

6. >& operator works appropriately? [4%]

```
$ 5>&1   sh -c 'echo ERROR! >&5'
ERROR!
```

7. Multiple redirections work appropriately? [4%]

```
$ >testfile 5>&1 6>&2  sh -c 'echo HELLO >&5; echo WORLD >&6'
WORLD
$ cat testfile
HELLO
```

8.5 Pipelines [6%]

1. Pipelines work appropriately? [3%]

```
$ echo hello world! | sed 's/hello/goodbye/' | cat -v
goodbye world!
```

2. Pipelines work with redirects? [3%]

```
$ 5>&1  sh -c 'echo hello world! >&5' | sed 's/hello/goodbye/' | cat -v
goodbye world!
```

8.6 Synchronous Commands [15%]

1. Synchronous commands run in foreground (BigShell waits on them)? [5%]

```
$ sleep 100
                # ...waiting...
```

2. Sending stop signal to synchronous commands causes them to be stopped and placed in the background? [5%]

```
$ sh -c 'sleep 5; killall -SIGSTOP sleep;' &
[0] 29347
$ sleep 100
                # ... 5 seconds later ...
[1] Stopped
[0] Done
$
```

3. Sending kill signal to synchronous commands causes them to exit, and `$?` is updated appropriately? [5%]

```
$ sh -c 'sleep 5; killall -SIGKILL sleep;' &
[0] 23959
$ sleep 100
                # ... 5 seconds later ...
[0] Done
$ echo $?
137
                # 137 = 128 + 9 (SIGKILL is signal # 9)
```

8.7 Signals [6%]

1. BigShell ignores the `SIGTSTP`, `SIGINT`, and `SIGTTOU` signals? [3%]

```
$ kill -s SIGTSTP $$
$ kill -s SIGINT $$
$ kill -s SIGTTOU $$
$
```

2. Child processes don't ignore these signals? [3%]

```
$ sleep 100
^C$           # ctrl-C -- SIGINT
$
```

8.8 Job Control

Entirely optional–an extra exercise if you want to push your understanding, it won't be tested in the grading script.