1.

In this stage, you'll implement support for printing error messages for invalid commands.

Example:

$ xyz

xyz: command not found

For now, we'll treat all commands as "invalid". In later stages we'll handle executing "valid" commands like echo, cd etc.

2.

In this stage, you'll implement a **REPL (Read-Eval-Print Loop)**.

The REPL

A **REPL (Read-Eval-Print Loop)** is an interactive loop that forms the core of a shell. It follows a repeating cycle:

1. **Read**: Display a prompt and wait for user input
2. **Eval**: Parse and execute the command
3. **Print**: Display the output or error message
4. **Loop**: Return to step 1 and wait for the next command

This cycle continues indefinitely until the shell process is terminated.

Your shell should follow this same cycle:

1. Display the prompt $, then wait for a line of input.
2. Print <command_name>: command not found for any command the user enters, like with the previous stages.
3. Return to step 1.

For example, if the user types hello, your shell should print hello: command not found, then display the prompt ($) again.

3.

In this stage, you'll implement the exit builtin.

The exit **Builtin**

The exit builtin is a special command that terminates the shell.

When your shell receives the exit command, it should terminate immediately.

4.

In this stage, you'll implement the echo builtin.

The echo **Builtin**

The echo builtin prints its arguments to stdout, with spaces between them, and a newline (₩n) at the end.

Example usage:

```
$ echo hello world

hello world

$ echo one two three

one two three
```

5.

In this stage, you'll implement the type builtin for your shell.

The type **Builtin**

The type builtin is used to determine how a command would be interpreted if it were used. It checks whether a command is a builtin, an executable file, or unrecognized.

For example:

```
$ type echo

echo is a shell builtin
```

$ type exit

exit is a shell builtin

$ type invalid_command

invalid_command: not found

For this stage, you'll handle two cases:

- For builtin commands (like echo, exit, and type), print <command> is a shell builtin.
- For unrecognized commands that don't match any builtin, print <command>: not found.

We'll handle executable files in later stages.

6.

In this stage, you'll extend the type builtin to search for executable files using PATH.

The PATH Environment Variable

The **PATH** environment variable specifies a list of directories where the shell should look for executable programs.

For example, if the PATH is set to /dir1:/dir2:/dir3, the shell would search for executables in /dir1, then /dir2, and finally /dir3, in that order.

Searching for Executables

When type receives a command input, your shell must follow these steps:

1. Check if the command is a builtin command (like exit or echo). If it is, report it as a builtin (<command> is a shell builtin) and stop.
2. If the command is not a builtin, your shell must go through every directory in PATH. For each directory:
    1. Check if a file with the command name exists.
    2. Check if the file has **execute permissions**.
    3. If the file exists and has execute permissions, print <command> is

        <full_path> and stop.

    4. If the file exists but **lacks execute permissions**, skip it and continue to the next directory.

3. If no executable is found in any directory, print <command>: not found.

For example:

$ type grep

grep is /usr/bin/grep

$ type invalid_command

invalid_command: not found

$ type echo

echo is a shell builtin

7.

In this stage, you'll add support for running external programs with arguments.

Running External Programs

So far, you've implemented builtin commands that your shell executes directly. Now you'll handle external programs that your shell needs to find and run.

When a command isn't a builtin, your shell should:

1. Search for an executable with the given name in the directories listed in PATH (just like type does)
2. If found, execute the program
3. Pass any arguments from the command line to the program

For example, if the user types custom_exe arg1 arg2, your shell should:

- Find custom_exe in PATH
- Execute it with three arguments: custom_exe (the program name), arg1, and arg2

8.

In this stage, you'll implement the pwd builtin.

The pwd **Builtin**

The pwd (print working directory) builtin prints the full, absolute path of the current working directory to stdout.

When your shell starts, its current working directory is typically the directory from which it was executed. Your pwd implementation needs to retrieve this information from the operating system and print it.

For example:

```
$ pwd

/home/user/projects

$ pwd

/usr/local/bin
```

9.

In this stage, you'll implement the cd builtin to handle absolute paths.

The cd **Builtin**

The cd (change directory) builtin is used to change the current working directory.

The cd command can handle different types of arguments:

- Absolute paths, like /usr/local/bin.
- Relative paths, like ./, ../, ./dir.
- The ~ character, which represents the user's home directory.

For this stage, we'll focus on absolute paths.

Handling Absolute Paths

An absolute path starts with / and specifies a location from the root of the filesystem.

When cd receives an absolute path:

- If the directory exists, change to that directory.
- If the directory doesn't exist, print cd: <directory>: No such file or directory.

For example:

```
$ pwd
/home/user
$ cd /usr/local/bin
$ pwd
/usr/local/bin
$ cd /does_not_exist
cd: /does_not_exist: No such file or directory
$ pwd
/usr/local/bin
```

If the directory change fails, the current directory should remain unchanged.

10.

In this stage, you'll extend your cd builtin to handle relative paths.

The cd **Builtin (Recap)**

As a recap, cd can receive multiple argument types:

- Absolute paths, like /usr/local/bin. (Handled in the previous stage)
- Relative paths, like ./, ../, ./dir.
- The ~ character.

For this stage, you'll handle the second argument type.

Relative Paths

A relative path specifies a location relative to the current working directory, rather than from the root of the filesystem.

Your shell must correctly interpret and navigate the following components of a relative path:

- ./ (Current Directory): Refers to the current working directory itself.
- ../ (Parent Directory): Refers to the directory immediately above the current working directory in the file system hierarchy.
- Subdirectories: Paths like ./dirname or dirname are treated as relative to the current directory.

Here are some examples:

```
$ pwd

/usr

$ cd ./local/bin      # Go to "local/bin" inside current directory (/usr)

$ pwd

/usr/local/bin

$ cd ../../           # Go up two levels

$ pwd

/usr

$ cd local            # "local" is shorthand for "./local"

$ pwd

/usr/local
```

11.

In this stage, you'll extend your cd builtin to handle the ~ character.

The cd **Builtin (Recap)**

As a recap, cd can receive multiple argument types:

- Absolute paths, like /usr/local/bin. (Handled in an earlier stage)
- Relative paths, like ./, ../, ./dir. (Handled in the previous stage)

- The ~ character.

The ~ Character

The ~ (tilde) character is shorthand for the user's home directory. It's a convenient way to quickly navigate back to your home directory from anywhere in the filesystem.

The home directory is specified by the HOME environment variable. When your shell sees cd ~, it should:

1. Read the value of the HOME environment variable
2. Change to that directory

For example:

```
$ pwd

/usr/local/bin

$ cd ~

$ pwd

/home/user

$ cd /var/log

$ pwd

/var/log

$ cd ~

$ pwd

/home/user
```

12.

In this stage, you'll implement support for quoting with single quotes.

Single Quotes

Single quotes (') disable all special meaning for characters enclosed within them. Every character between single quotes is treated literally.

When your shell parses a command line:

- Characters inside single quotes (including escape characters and potential special characters like $, *, or ~) lose their special meaning and are treated as normal characters.
- Consecutive whitespace characters (spaces, tabs) inside single quotes are preserved and are not collapsed or used as delimiters.
- Quoted strings placed next to each other are concatenated to form a single argument.

Here are a few examples illustrating how single quotes behave:

Command	**Expected output**	**Explanation**
echo 'hello world'	hello world	Spaces are preserved within quotes.
echo hello world	hello world	Consecutive spaces are collapsed unless quoted.
echo 'hello''world'	helloworld	Adjacent quoted strings 'hello' and 'world' are concatenated.
echo hello''world	helloworld	Empty quotes '' are ignored.

13.

In this stage, you'll implement support for quoting with double quotes.

Double Quotes

In shell syntax, most characters within **double quotes** (") are treated literally. However, double quotes allow certain special characters to be interpreted (like $ for variables and ₩ for escaping), but we'll cover those exceptions in later stages.

For this stage, your shell must apply the following rules when parsing double quotes:

- Consecutive whitespaces (spaces, tabs) must be preserved.
- Characters that normally act as delimiters or special characters lose their special meaning inside double quotes and are treated literally.
- Double-quoted strings placed next to each other are concatenated to form a single argument.

For example:

```
$ echo "hello     world"
hello     world           # Multiple spaces preserved


$ echo "hello""world"
helloworld                # Quoted strings next to each other are concatenated.


$ echo "hello" "world"
hello world               # Separate arguments


$ echo "shell's test"
shell's test              # Single quotes inside are literal
```

14.

In this stage, you'll implement support for backslashes outside quotes.

Backslash Escaping

When a backslash (₩) is used outside of quotes, it acts as an escape character. If the backslash precedes a character that usually has a special meaning to the shell (like $, *, ?, or other delimiters), the backslash causes the character to be treated as a literal character.

Here are a few examples illustrating how backslashes behave outside quotes:

Command	Expected Output	Explanation
echo world₩ ₩ ₩ ₩ ₩ ₩ script	world script	Each ₩ creates a literal space as part of one argument.
echo before₩ after	before after	The backslash preserves the first space literally, but the shell collapses the subsequent unescaped spaces.
echo hello₩₩world	hello₩world	The first backslash escapes the second, and the result is a single literal backslash in the argument.

15.

In this stage, you'll implement support for backslashes within single quotes.

Backslashes in Single Quotes

Backslashes have no special escaping behavior inside **single quotes**. Every character (including backslashes) within single quotes is treated literally.

For example:

```
$ echo 'shell₩₩₩nscript'

shell₩₩₩nscript


$ echo 'example₩"test'

example₩"test
```

16.

In this stage, you'll implement support for backslashes within double quotes.

Backslashes in Double Quotes

Within **double quotes**, a backslash only escapes certain special characters: ", ₩, $, `, and newline. For all other characters, the backslash is treated literally.

In this stage, we'll cover:

- ₩": escapes double quote, allowing " to appear literally within the quoted string.
- ₩₩: escapes backslash, resulting in a literal ₩.

We won't cover the following cases in this stage:

- ₩$: escapes the dollar sign.
- ₩`: escapes the backtick.
- ₩<newline>: escapes a newline character.

Here are a few examples illustrating how backslashes behave within double quotes:

Command	**Expected output**
echo "A ₩₩ escapes itself"	A ₩ escapes itself
echo "A ₩" inside double quotes"	A " inside double quotes

17.

In this stage, you'll implement support for executing a quoted executable.

Quoted Executable Names

Up until now, the executable name (the first word in a command) has been a simple string. But executable names can also be quoted, just like arguments. This is useful when an executable has spaces, quotes, or special characters in its name.

Quoting the executable name lets you execute programs that would otherwise be difficult to reference.

For example:

```
$ 'my program' argument1
```

# Runs an executable named "my program"

$ "exe with spaces" file.txt

# Runs an executable named "exe with spaces"

When your shell receives a command with a quoted executable:

- Your parser must correctly interpret the quotes to form the executable name (e.g., from "my 'program'" to my 'program').
- Your shell must search for the unquoted program name in the PATH directories.
- Upon finding the executable, your shell must spawn the process and pass any other tokens as arguments.

18.

In this stage, you'll implement support for redirecting the output of a command to a file.

The 1> operator is used to redirect the output of a command to a file. But, as a special case, if the file descriptor is not specified before the operator >, the output is redirected to the standard output by default, so > is equivalent to 1>.

Learn more about **Redirecting Output**.

19.

In this stage, you'll implement support for redirecting the standard error of a command to a file.

The 2> operator is used to redirect the standard error of a command to a file.

Learn more about **Redirecting Stderr**.