# The Bash Parser

It is imperative that you have a good understanding of how Bash reads your commands in and parses them into executable code. Knowing how Bash works with your code is the key to writing code that works well with Bash.

**Note:** Chet Ramey's chapter from *The Architecture of Open Source Applications* also describes the Bash parser in detail.

- **Step 1: Read data to execute.**

  Bash always reads your script or commands on the bash command prompt *line by line*. If your line ends with a backslash character, bash reads another line before processing the command and appends that other line to the current, with a literal newline inbetween. *(I will from here on refer to the chunk of data Bash read in as the **line** of data; even though it is technically possible that this line contains one or more newlines.)*

  **Step Input:**
  `echo "What's your name?"`
  `read name; echo "$name"`

  **Step Output:**
  `echo "What's your name?"`
  *and*
  `read name; echo "$name"`

- **Step 2: Process quotes.**

  Once Bash has read in your line of data, it looks through it in search of quotes. The first quote it finds triggers a quoted state for all characters that follow up until the next quote of the same type. If the quoted state was triggered by a double quote (`"..."`), all characters except for `$`, `"`, `` ` `` and `\` lose any special meaning they might have. That includes single quotes, spaces and newlines, etc. If the quoted state was triggered by a single quote (`'...'`), all characters except for `'` lose their special meaning. Yes, also `$` and `\`. Therefore, the following command will produce literal output:

```
 $ echo 'Back\Slash $dollar "Quote"'
Back\Slash $dollar "Quote"
```

  The fact that the backslash loses its ability to cancel out the meaning of the next character means that this will not work:

```
 $ echo 'Don\'t do this'
 >
```

  Bash will ask you for the next line of input because unlike what we *thought* we did, the **second** quote, the one we tried to escape with the backslash, actually **closed our quoted state** meaning the `t do this` was **not** quoted. The last quote on the line then **opened** our quoted state again, and bash asks for more input until it is closed again (it tries to finish step 1: it reads data until it finds an unescaped newline. The opened single quote state is escaping our newline). Now that bash knows which of the characters in the line of data are escaped (stripped of their ability to mean anything special to Bash) and which are not, Bash removes the quotes that were used to determine this from the data and proceeds to the next step.

  **Step Input:**
  `echo "What's your name?"`

  **Step Output:**
  `echo` *`What's your name?`*

  (**Note:** Every character originally between the double quotes has been marked as *escaped*. I will mark escaped characters in these examples by making them *`italic`*.)

- **Step 3: Split the read data into commands.**

  Our line is now split up into separate commands using `;` as a command separator. Remember from the previous step that any `;` characters that were quoted or escaped do not have their special meaning anymore and will not be used for command splitting. They will just appear in the resulting command line literally:

```
 $ echo "What a lovely day; and sunny, too!"
What a lovely day; and sunny, too!
```

  **Step Input:**
  `read name; echo` *`$name`*

  **Step Output:**
  `read name`
  *and*

```
echo $name
```

The following steps are executed for each command that resulted from splitting up the line of data:

- **Step 4: Parse special operators.**

  Look through the command to see whether there are any special operators such as `{..}`, `<(..)`, `< ...`, `<<< ..`, `.. | ..`, etc. These are all processed in a specific order. Redirection operators are removed from the command line, other operators are replaced by their resulting expression (eg. `{a..c}` is replaced by `a b c`).

  **Step Input:**
  ```
  diff <(foo) <(bar)
  ```

  **Step Output:**
  ```
  diff /dev/fd/63 /dev/fd/62
  ```

  (**Note:** The `<(..)` operator starts a background process to execute the command `foo` (and one for `bar`, too) and sends the output to a file. It then replaces itself with the pathname of that file.)

- **Step 5: Perform Expansions.**

  Bash has many operators that involve expansion. The simplest of these is `$parameter`. The dollar sign followed by the name of a parameter, which optionally might be surrounded by braces, is called *Parameter Expansion*. What Bash does here is basically just replace the *Parameter Expansion* operator with the contents of that parameter. As such, the command `echo $USER` will in this step be converted to `echo lhunath` with me. Other expansions include *Pathname Expansion* (`echo *.txt`), *Command Substitution* (`rm "$(which nano)"`), etc.

  **Step Input:**
  ```
  echo "$PWD has these files that match *.txt :" *.txt
  ```

  **Step Output:**
  ```
  echo /home/lhunath/docs has these files that match *.txt : bar.txt foo.txt
  ```

- **Step 6: Split the command into a command name and arguments.**

  The name of the command Bash has to execute is always the **first word** in the line. The rest of the command data is split into words which make the arguments. This process is called *Word Splitting*. Bash basically cuts the command line into pieces wherever it sees whitespace. This whitespace is completely removed and the pieces are called *words*. Whitespace in this context means: Any spaces, tabs or newlines that are **not escaped**. (Escaped spaces, such as spaces inside quotes, lose their special meaning of whitespace and are not used for splitting up the command line. They appear literally in the resulting arguments.) As such, if the name of the command that you want to execute or one of the arguments you want to pass contains spaces that you don't want bash to use for cutting the command line into words, you can use quotes or the backslash character:

  ```
     My Command /foo/bar   ## This will execute the command named 'My' because it is the first word.
     "My Command" /foo/bar ## This will execute the command named 'My Command' because the space inside
  the quotes has lost its special meaning allowing it to split words.
  ```

  **Step Input:**
  ```
  echo "/home/lhunath/docs has these files that match *.txt :" bar.txt foo.txt
  ```

  **Step Output:**
  **Command Name:** '`echo`'
  **Argument 1:** '`/home/lhunath/docs has these files that match *.txt :`'
  **Argument 2:** '`bar.txt`'
  **Argument 3:** '`foo.txt`'

- **Step 7: Execute the command.**

  Now that the command has been parsed into a command name and a set of arguments, Bash executes the command and sets the command's arguments to the list of words it has generated in the previous step. If the command type is a function or builtin, the command is executed by the same Bash process that just went through all these steps. Otherwise, Bash will first fork off (create a new bash process), initialize the new bash processes with the settings that were parsed out of this command (redirections, arguments, etc.) and execute the command in the forked off bash process (child process). The parent (the Bash that did these steps) waits for the child to complete the command.

  **Step Input:**
  ```
  sleep 5
  ```

  **Causes:**
  ```
  ├┬· 33321 lhunath -bash
  │├──· 46931 lhunath sleep 5
  ```

After these steps, the next command, or next line is processed. Once the end of the file is reached (end of the script or the interactive bash session is closed) bash stops and returns the exit code of the last command it has executed.

### Graphical Example

For a more simplified example of the process, see: http://stuff.lhunath.com/parser.png

### Common Mistakes

These steps might seem like common sense after looking at them closely, but they can often seem counter-intuitive for certain specific cases. As an example, let me enumerate a few cases where people have often made mistakes against the way they think bash will interpret their command:

- `start=1; end=5; for number in {$start..$end}`: *Brace Expansion* happens in step 4, while *Parameter Expansion* happens in step 5. *Brace Expansion* tries to expand `{$start..$end}` but can't. It sees the `$start` and `$end` as strings, not *Parameter Expansion*s and gives up:

  Step 4 Results:

  ```
  start=1
  end=5
  for number in {$start..$end}
  ```

  Step 5 Results:

  ```
  start=1
  end=5
  for number in {1..5}
  ```

  And `number` will now become `{1..5}` instead of `1`. No *Brace Expansion* has been performed.

- `[ $name = B. Foo ]`: *Word Splitting* will break this example. The `test` program (`[`) looks for four arguments in this case. A left hand side, an operator, a right hand side, and a closing `]`. To find out what's wrong with this command, do as Bash does: Chop the command up into arguments. Assuming `name` contains `B. Foo`:
  - `[`
  - `B.`
  - `Foo`
  - `=`
  - `B.`
  - `Foo`
  - `]`

  A whole lot more than four. You need to use *Quotes* to prevent the space between `B.` and `Foo` from causing *Word Splitting*. Quote the `B. Foo` **AND** the `$name` so that when `$name` is expanded, the whitespace in `B. Foo` is treated the same as on the right hand side. It is important to remember that step 5 (*Perform Expansion*) comes **before** step 6 (*Split the command into a command name and arguments*). That means that `$name` is not safe from having its result cut up, because the cutting up happens after `$name` is replaced by the value within `name`.

CategoryShell

BashParser (last edited 2017-10-10 14:25:02 by vrsimsbury)