Basic grammar rules of Bash

Bash builds its features on top of a few basic **grammar rules**. The code you see everywhere, the code you use, is based on those rules. However, **this is a very theoretical view**, but if you're interested, it may help you understand why things look the way they look.

If you don't know the commands used in the following examples, just trust the explanation.

Simple Commands

Bash manual says:

```
A simple command is a sequence of optional variable assignments followed by blank-separated words and redirections,
and terminated by a control operator. The first word specifies the command to be executed, and is passed as argument
zero.  The remaining words are passed as arguments to the invoked command.
```

Sounds harder than it actually is. It is what you do daily. You enter simple commands with parameters, and the shell executes them.

Every complex Bash operation can be split into simple commands:

```
ls
ls > list.txt
ls -l
LC_ALL=C ls
```

The last one might not be familiar. That one simply adds " `LC_ALL=C` " to the environment of the `ls` program. It doesn't affect your current shell. This also works while calling functions, unless Bash runs in POSIX® mode (in which case it affects your current shell).

Every command has an exit code. It's a type of return status. The shell can catch it and act on it. Exit code range is from 0 to 255, where 0 means success, and the rest mean either something failed, or there is an issue to report back to the calling program.

The simple command construct is the **base** for all higher constructs. Everything you execute, from pipelines to functions, finally ends up in (many) simple commands. That's why Bash only has one method to expand and execute a simple command.

Pipelines

Fix Me! Missing an additional article about pipelines and pipelining

`[time [-p]] [ ! ] command [ | command2 … ]`

Don't get confused about the name "pipeline." It's a grammatic name for a construct. Such a pipeline isn't necessarily a pair of commands where stdout/stdin is connected via a real pipe.

Pipelines are one or more **simple commands** (separated by the `|` symbol connects their input and output), for example:

```
ls /etc | wc -l
```

will execute `ls` on `/etc` and **pipe** the output to `wc`, which will count the lines generated by the ls command. The result is the number of directory entries in /etc.

The last command in the pipeline will set the exit code for the pipeline. This exit code can be "inverted" by prefixing an exclamation mark to the pipeline: An unsuccessful pipeline will exit "successful" and vice versa. In this example, the commands in the if stanza will be executed if the pattern "^root:" is **not** found in `/etc/passwd`:

```
if ! grep '^root:' /etc/passwd; then
  echo "No root user defined... eh?"
fi
```

Yes, this is also a pipeline (although there is no pipe!), because the **exclamation mark to invert the exit code** can only be used in a pipeline. If `grep`'s exit code is 1 (FALSE) (the text was not found), the leading `!` will "invert" the exit code, and the shell sees (and acts on) exit code 0 (TRUE) and the `then` part of the `if` stanza is executed. One could say we checked for "`not grep "^root" /etc/passwd`".

The set option pipefail determines the behavior of how bash reports the exit code of a pipeline. If it's set, then the exit code (`$?`) is the last command that exits with non zero status, if none fail, it's zero. If it's not set, then `$?` always holds the exit code of the last command (as explained above).

The shell option `lastpipe` will execute the last element in a pipeline construct in the current shell environment, i.e. not a subshell.

There's also an array `PIPESTATUS[]` that is set after a foreground pipeline is executed. Each element of `PIPESTATUS[]` reports the exit code of the respective command in the pipeline. Note: (1) it's only for foreground pipe and (2) for higher level structure that is built up from a pipeline. Like list, `PIPESTATUS[]` holds the exit status of the last pipeline command executed.

Another thing you can do with pipelines is log their execution time. Note that **`time` is not a command**, it is part of the pipeline syntax:

```
# time updatedb
real    3m21.288s
user    0m3.114s
sys     0m4.744s
```

Lists

Fix Me! Missing an additional article about list operators

A list is a sequence of one or more **pipelines** separated by one of the operators `;` , `&` , `&&` , or `||` , and optionally terminated by one of `;` , `&` , or `<newline>` .

⇒ It's a group of **pipelines** separated or terminated by **tokens** that all have **different meanings** for Bash.

Your whole Bash script technically is one big single list!

Operator	Description
`<PIPELINE1>` **`<newline>`** `<PIPELINE2>`	Newlines completely separate pipelines. The next pipeline is executed without any checks. (You enter a command and press `<RETURN>` !)
`<PIPELINE1>` **`;`** `<PIPELINE2>`	The semicolon does what `<newline>` does: It separates the pipelines
`<PIPELINE>` **`&`** `<PIPELINE>`	The pipeline in front of the `&` is executed **asynchronously** ("in the background"). If a pipeline follows this, it is executed immediately after the async pipeline starts
`<PIPELINE1>` **`&&`** `<PIPELINE2>`	`<PIPELINE1>` is executed and **only** if its exit code was 0 (TRUE), then `<PIPELINE2>` is executed (AND-List)
`<PIPELINE1>` **`||`** `<PIPELINE2>`	`<PIPELINE1>` is executed and **only** if its exit code was **not** 0 (FALSE), then `<PIPELINE2>` is executed (OR-List)

Note: POSIX calls this construct a "compound lists".

Compound Commands

See also the list of compound commands.

There are two forms of compound commands:

- form a new syntax element using a list as a "body"
- completly independant syntax elements

Essentially, everything else that's not described in this article. Compound commands have the following characteristics:

- they **begin** and **end** with a specific keyword or operator (e.g. `for … done`)
- they can be redirected as a whole

See the following table for a short overview (no details - just an overview):

Compound command syntax	Description
`( <LIST> )`	Execute `<LIST>` in an extra subshell ⇒ article

Compound command syntax	Description
`{ <LIST> ; }`	Execute `<LIST>` as separate group (but not in a subshell) ⇒ article
`(( <EXPRESSION> ))`	Evaluate the arithmetic expression `<EXPRESSION>` ⇒ article
`[[ <EXPRESSION> ]]`	Evaluate the conditional expression `<EXPRESSION>` (aka "the new test command") ⇒ article
`for <NAME> in <WORDS> ; do <LIST> ; done`	Executes `<LIST>` while setting the variable `<NAME>` to one of `<WORDS>` on every iteration (classic for-loop) ⇒ article
`for (( <EXPR1> ; <EXPR2> ; <EXPR3> )) ; do <LIST> ; done`	C-style for-loop (driven by arithmetic expressions) ⇒ article
`select <NAME> in <WORDS> ; do <LIST> ; done`	Provides simple menus ⇒ article
`case <WORD> in <PATTERN>) <LIST> ;; … esac`	Decisions based on pattern matching - executing `<LIST>` on match ⇒ article
`if <LIST> ; then <LIST> ; else <LIST> ; fi`	The if clause: makes decisions based on exit codes ⇒ article
`while <LIST1> ; do <LIST2> ; done`	Execute `<LIST2>` while `<LIST1>` returns TRUE (exit code) ⇒ article
`until <LIST1> ; do <LIST2> ; done`	Execute `<LIST2>` until `<LIST1>` returns TRUE (exit code) ⇒ article

Shell Function Definitions

Fix Me! Missing an additional article about shell functions

A shell function definition makes a **compound command** available via a new name. When the function runs, it has its own "private" set of positional parameters and I/O descriptors. It acts like a script-within-the-script. Simply stated: **You've created a new command.**

The definition is easy (one of many possibilities):

`<NAME> () <COMPOUND_COMMAND> <REDIRECTIONS>`

which is usually used with the `{…; }` compound command, and thus looks like:

```
print_help() { echo "Sorry, no help available"; }
```

As above, a function definition can have any **compound command** as a body. Structures like

```
countme() for ((x=1;x<=9;x++)); do echo $x; done
```

are unusual, but perfectly valid, since the for loop construct is a compound command!

If **redirection** is specified, the redirection is not performed when the function is defined. It is performed when the function runs:

```
# this will NOT perform the redirection (at definition time)
f() { echo ok ; } > file

# NOW the redirection will be performed (during EXECUTION of the func
tion)
f
```

Bash allows three equivalent forms of the function definition:

```
NAME ()          <COMPOUND_COMMAND> <REDIRECTIONS>
function NAME () <COMPOUND_COMMAND> <REDIRECTIONS>
function NAME    <COMPOUND_COMMAND> <REDIRECTIONS>
```

The space between `NAME` and `()` is optional, usually you see it without the space.

I suggest using the first form. It's specified in POSIX and all Bourne-like shells seem to support it.

Note: Before version `2.05-alpha1`, Bash only recognized the definition using curly braces (`name() { … }`), other shells allow the definition using **any** command (not just the compound command set).

To execute a function like a regular shell script you put it together like this:

```
#!/bin/bash
# Add shebang

mycmd()
{
  # this $1 belongs to the function!
  find / -iname "$1"
}

# this $1 belongs the script itself!
mycmd "$1" # Execute command immediately after defining function

exit 0
```

Just informational(1):

Internally, for forking, Bash stores function definitions in environment variables. Variables with the content "*() ….*".

Something similar to the following works without "officially" declaring a function:

```
$ export testfn="() { echo test; }"
$ bash -c testfn
test
$
```

Just informational(2):

It is possible to create function names containing slashes:

```
/bin/ls() {
  echo LS FAKE
}
```

The elements of this name aren't subject to a path search.

Weird function names should not be used. Quote from the maintainer:

- *It was a mistake to allow such characters in function names (`unset' doesn't work to unset them without forcing -f, for instance). We're stuck with them for backwards compatibility, but I don't have to encourage their use.*

Grammar summary

- a **simple command** is just a command and its arguments
- a **pipeline** is one or more **simple command** probably connected in a pipe
- a **list** is one or more **pipelines** connected by special operators
- a **compound command** is a **list** or a special command that forms a new meta-command
- a **function definition** makes a **compound command** available under a new name, and a separate environment

Examples for classification

Fix Me! more…

A (very) simple command

```
echo "Hello world..."
```

All of the following are simple commands

```
x=5
```

```
>tmpfile
```

```
{x}<"$x" _=${x=<(echo moo)} <&0$(cat <&"$x" >&2)
```

A common compound command

```
if [ -d /data/mp3 ]; then
  cp mymusic.mp3 /data/mp3
fi
```

- the **compound command** for the `if` clause
- the **list** that `if` **checks** actually contains the **simple command** `[ -d /data/mp3 ]`
- the **list** that `if` **executes** contains a simple command (`cp mymusic.mp3 /data/mp3`)

Let's invert test command exit code, only one thing changes:

```
if ! [ -d /data/mp3 ]; then
  cp mymusic.mp3 /data/mp3
fi
```

- the **list** that `if` **checks** contains a **pipeline** now (because of the `!`)

See also

- Internal: List of compound commands
- Internal: Parsing and execution of simple commands
- Internal: Quoting and escaping
- Internal: Introduction to expansions and substitutions
- Internal: Some words about words...

Discussion

Jacob, 2010/12/28 01:43 (), 2010/12/28 06:50 ()

"Basically, pipelines are one or more simple commands"

Actually, the bash man page defines "simple commands" and "compound commands" explicitly, but does not define (plain old) "commands". Presumably they can be simple commands or compound commands. Witness:

```
for NUM in {1..100}; do echo "$NUM"; done | fgrep 3 | while read
NUM; do echo "${NUM/3/three}"; done
```

The REDIRECTION section also implies that "command" includes "compound commands" by the phrase "The following redirection operators may precede or appear anywhere within a simple command or may follow a command." For example:

```
while read -a LIST; do echo "${LIST[@]}"; done < <(sed "s/#.*//"
/etc/fstab | grep .)
```

Jan Schampera, 2010/12/28 06:49 ()

Yes, I know the glitch here. `command` is defined as

- simple command
- compound command
- function definition
- (Bash) coproc definition

Bash manpage is a bit unlucky here (and this page, too!).

I will make some thoughts about it, the best would be an "advanced syntax description" page that includes (simplified) syntax diagrams based on incomplete (not down to every string and character definition) EBNF to be more accurate. It's just some amount of work and will have mistakes at first.

syntax/basicgrammar.txt Last modified: 2019/04/01 21:45 by ddebhw