Navigating Linux System Commands

A guide for beginners to the Shell and GNU coreutils

Sayan Ghosh

August 1, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on August 1, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

Preface **v**

Contents **vii**

1 Shell Scripting **1**
1.1 What is a shell script? . . . 1
1.2 Shebang . . . 1
1.3 Comments . . . 2
1.3.1 Multiline Comments . . . 2
1.4 Variables . . . 3
1.5 Arguments . . . 4
1.5.1 Shifting Arguments . . . 6
1.6 Input and Output . . . 7
1.6.1 Reading Input . . . 7
1.7 Conditionals . . . 8
1.7.1 Test command . . . 8
1.7.2 Test Keyword . . . 11
1.8 If-elif-else . . . 13
1.8.1 If . . . 13
1.8.2 Else . . . 14
1.8.3 Elif . . . 14
1.9 Exit code inversion . . . 15
1.10 Mathematical Expressions as if command . . . 15
1.11 Command Substitution in if . . . 15
1.12 Switch . . . 15
1.12.1 Fall Through . . . 16
1.12.2 Multiple Patterns . . . 16
1.13 Select Loop . . . 17
1.14 Loops . . . 18
1.14.1 For loop . . . 18
1.14.2 C style for loop . . . 20
1.14.3 IFS . . . 21
1.14.4 While loop . . . 22
1.14.5 Until loop . . . 23
1.14.6 Read in while . . . 24
1.14.7 Break and Continue . . . 25
1.15 Functions . . . 27
1.16 Debugging . . . 29
1.17 Recursion . . . 30
1.18 Shell Arithmetic . . . 31
1.18.1 bc . . . 32
1.18.2 expr . . . 32
1.19 Running arbritary commands using source, eval and exec . . . 34
1.19.1 exec . . . 34
1.20 Getopts . . . 34
1.20.1 With case statement . . . 36

1.21 Profile and RC files 36
1.22 Summary 38

List of Figures

1.1 Flowchart of the `if`, `elif`, and `else` construct 14

List of Tables

1.1 Differences between range expansion and seq command 20
1.2 Differences between seq and python's range function 20
1.3 Summary of the bash constructs 38

1 Shell Scripting

Now that we have learnt the basics of the bash shell and the core utils, we will now see how we can combine them to create shell scripts which let us do even more complex tasks in just one command.

1.1 What is a shell script?

A shell script is a file that contains a sequence of shell commands. When you run a shell script, the commands in the script are executed in the order they are written. Shell scripts are used to automate tasks that would otherwise require a lot of manual work. They are also used to create complex programs that can be run from the command line.

1.2 Shebang

Definition 1.2.1 (Shebang) The shebang is a special line at the beginning of a shell script that tells the operating system which interpreter to use to run the script. The shebang is written as `#!` followed by the path to the interpreter.

For example, the shebang for a bash script is `#!/bin/bash`. However, as `/bin` is a symbolic link to `/usr/bin` in most systems, we can also specify the path as `#!/usr/bin/bash`

The shebang is only needed if we run the script directly from the shell, without specifying the interpreter. This also requires the script to have the execute permission set.

```
$ cat script.sh
#!/bin/bash
echo "Hello, World!"
$ bash script.sh
Hello, World!
$ ./script.sh
bash: ./script.sh: Permission denied
$ chmod +x script.sh
$ ./script.sh
Hello, World!
```

Here we are first running the script with the `bash` command, which is why the shebang and execute permission is not needed. However, when we try to run the script directly, we get a permission denied error as the permission is not set. We then set the execute permission and run the script again, which works.

We can also use the shebang for non-bash scripts, for example, a python script can have the shebang `#!/usr/bin/python3`.

1.1 What is a shell script? . 1
1.2 Shebang 1
1.3 Comments 2
1.3.1 Multiline Comments . . 2
1.4 Variables 3
1.5 Arguments 4
1.5.1 Shifting Arguments . . . 6
1.6 Input and Output 7
1.6.1 Reading Input 7
1.7 Conditionals 8
1.7.1 Test command 8
1.7.2 Test Keyword 11
1.8 If-elif-else 13
1.8.1 If 13
1.8.2 Else 14
1.8.3 Elif 14
1.9 Exit code inversion . . . 15
1.10 Mathematical Expressions as if command . . 15
1.11 Command Substitution in if 15
1.12 Switch 15
1.12.1 Fall Through 16
1.12.2 Multiple Patterns 16
1.13 Select Loop 17
1.14 Loops 18
1.14.1 For loop 18
1.14.2 C style for loop 20
1.14.3 IFS 21
1.14.4 While loop 22
1.14.5 Until loop 23
1.14.6 Read in while 24
1.14.7 Break and Continue . . . 25
1.15 Functions 27
1.16 Debugging 29
1.17 Recursion 30
1.18 Shell Arithmetic 31
1.18.1 bc 32
1.18.2 expr 32
1.19 Running arbritary commands using source, eval and exec 34
1.19.1 exec 34
1.20 Getopts 34
1.20.1 With case statement . . . 36
1.21 Profile and RC files . . . 36
1.22 Summary 38

```
$ cat script.py
#!/usr/bin/python3
print("Hello, World!")
$ python3 script.py
Hello, World!
$ chmod u+x script.py
$ ./script.py
Hello, World!
```

Even though we called the script without specifying the interpreter, the shebang tells the shell to use the python3 interpreter to run the script. This only works if the file has the execute permission set.

If the shebang is absent, and a file is executed without specifying the interpreter, the shell will try to execute it in its own interpreter.

1.3 Comments

In shell scripts, comments are lines that are not executed by the shell. Comments are used to document the script and explain what the script does. Comments in shell scripts start with a # character and continue to the end of the line.

Although it may not be required when running a command from the terminal, comments are supported there as well. Comments are most useful in scripts, where they can be used to explain what the script does and how it works.

```
$ echo "hello" # This is a comment
$ echo "world" # we are using the echo command to print the word "
    world"
```

Here we use the # to start a line comment. Any character after the pound sign is ignored by the interpreter.

Remark 1.3.1 The shebang also starts with a # character, so it is also ignored by the interpreter when executing, however, if a file is executed without mentioning the interpreter then the shell reads its first line and find the path to the interpreter using the shebang

1.3.1 Multiline Comments

Although bash does not have a built-in syntax for multiline comments, we can use a trick to create multiline comments. We can use the NOP command : to create a multiline comment by passing the multiline string to it.

```
$ : '
This is a multiline comment
This is the second line
This is the third line
'
```

1.4 Variables

We have already seen the various ways to create and use variables in bash. We can use these variables in scripts as well.

```
$ cat variables.sh
#!/bin/bash
name="alice"
declare -i dob=2001
echo "Hello, ${name^}! You are $((2024 - dob)) years old."
$ chmod u+x variables.sh
$ ./variables.sh
Hello, Alice! You are 23 years old.
```

The variables defined inside a script are not available outside the script if the script is executed in a new environment. However, if we want to actually retain the variables, we can use the `source` command to run the script in the current environment.

This reads the script and executes it in the current shell, so the variables defined in the script are available in the current shell. The PID of the script remains same as the executing shell, and no new process is created. Since this only reads the file, the execute permission is not needed.

This is useful when we want to set environment variables or aliases using a script.

```
$ cat variables.sh
name="alice"
dob=2001
$ source variables.sh
$ echo "Hello, ${name^}! You are $((2024 - dob)) years old."
Hello, Alice! You are 23 years old.
```

Similarly, if we want to run a script in a new environment, but we still want it to have read access to the variables declared in the current environment, we can export the variables needed.

In this example we can see that if a variable is not defined in the scrtipt and we access the variable in the script, it will take an empty string. If we define the variable in the parent shell and then call the script, it will still not have access to it. However, if we export the variable, then calling the script will provide it with the variable.

```
$ cat variables.sh
#!/bin/bash
echo "Hello, ${name^}! You are $((2024 - dob)) years old."
$ chmod u+x variables.sh
$ ./variables.sh
Hello, ! You are 2024 years old.
$ name="alice"
$ dob=2001
$ ./variables.sh
Hello, ! You are 2024 years old.
$ export name dob
$ ./variables.sh
Hello, Alice! You are 23 years old.
```

This is because `export` makes the variable into an environment variable, which is available to all child processes of the current shell. When a new environment is created, the environment variables are copied to the new environment, so the script has access to the exported variables.

However, if we want to add some variable to the environment of the script, but we do not want to add it to our current shell's environment, we can directly specify the variable and the value before running the script,

in the same line. This sets the variables for the script's environment, but not for the shell's environment.

```
$ cat variables.sh
#!/bin/bash
echo "Hello, ${name^}! You are $((2024 - dob)) years old."
$ ./variables.sh
Hello, ! You are 2024 years old.
$ name=Alice dob=2001 ./script.sh
Hello, Alice! You are 23 years old.
```

1.5 Arguments

Just like we can provide arguments to a command, we can also provide arguments to a script. These arguments are stored in special variables that can be accessed inside the script.

- `$0` - The path of the script/interpreter by which it is called.
- `$1` - The first argument
- `$2` - The second argument

  ⋮
- `$@` - All arguments stored as an array
- `$*` - All arguments stored as a space separated string
- `$#` - The number of arguments

```
$ cat arguments.sh
echo $0
$ ./arguments.sh
./arguments.sh
$ bash arguments.sh
arguments.sh
$ source arguments.sh
/bin/bash
$ echo $0
/bin/bash
```

If we know the number of arguments that will be passed to the script, we can directly access them using the `$n` syntax.

```
$ cat arguments.sh
echo $1 $2
echo $3
$ bash arguments.sh Hello World "This is a multiword single
    argument"
Hello World
This is a multiword single argument
```

Here we can also see how to pass multiple words as a single argument, by quoting it.

Sometimes, we may want to pass the value of a variable as an argument, in those cases, we should always quote the variable expansion, as if the variable contains multiple words, bash will expand the variables and treat each word as a separate argument, instead of the entire value of the variable as a single argument.

Quoting the variable prevents word splitting.

```
$ cat split.sh
echo $1
$ var="Hello World"
$ bash split.sh $var
Hello
$ bash split.sh "$var"
Hello World
```

The first attempt prints only `Hello` as the second word is stored in `$2`. But if we quote the variable expansion, then the entire string is taken as one parameter.

Double vs Single Quotes

There are some subtle differences between double and single quotes. Although both are used to prevent word splitting, single quotes also prevent variable expansion, while double quotes do not.

```
$ echo "Hello $USER"
Hello sayan
$ echo 'Hello $USER'
Hello $USER
```

However, if we do not know the number of arguments that will be passed to the script, we can use the `$@` and `$*` variables to access all the arguments.

```
$ cat all-arguments.sh
echo "$@"
$ bash all-arguments.sh Hello World "This is a multiword single
    argument"
Hello World This is a multiword single argument
```

Here we can see that `$@` expands to all the arguments as an array, while `$*` expands to all the arguments as a single string. In this case, the output of both looks same as `echo` its arguments side by side, separated by a space.

We can observe the difference between `$@` and `$*` when we use them in a loop, or pass to any command that shows output for each argument on separate lines.

```
$ cat multiword.sh
touch "$@"
$ bash multiword.sh Hello World "This is a multiword single
    argument"
$ ls -1
Hello
multiword.sh
'This is a multiword single argument'
World
```

In this case, since we are using `"$@"` (the quoting is important), the `touch` command is called with each argument as a separate argument, so it creates a file for each argument. The last array element has multiple words, but it creates a single file with the entire sentence as the file name.

Whereas if we use `$*`, the arguments are stored as a string, and it is split by spaces, so the touch command will create a file for each word.

```
$ cat multiword.sh
touch $*
$ bash multiword.sh Hello World "This is a multiword single
    argument"
$ ls -1
a
argument
Hello
is
multiword
multiword.sh
single
This
World
```

In this case, the `touch` command is called with each word as a separate argument, so it creates a file for each word.

> **Exercise 1.5.1** Similarly, try out the other way of iterating over the arguments (`"$*"`) and observe the difference.

We can also iterate over the indices and access each argument using a for loop. To find the number of arguments, we can use `$#`.

```
$ cat args.sh
args=("$@")
echo $#
for ((i=0; i < $#; i++)); do
  echo "${args[i]}"
done
$ bash args.sh hello how are you
4
hello
how
are
you
```

Here we are using `$#` to get the number of arguments the script has recieved and then dynamically iterating that many times to access each element of the `args` array.

1.5.1 Shifting Arguments

Sometimes we may want to remove the first few arguments from the list of arguments. We can do this using the `shift` command.

`shift` is a shell builtin that shifts the arguments to the left by one. The first argument is removed, and the second argument becomes the first argument, the third argument becomes the second argument, and so on.

```
$ cat shift.sh
echo "First argument is $1"
shift
```

```
echo "Rest of the arguments are $*"
$ bash shift.sh one two three four
First argument is one
Rest of the arguments are two three four
```

1.6 Input and Output

Now that we have seen how to pass arguments to a script, we will see how to take input from the user and display output to the user.

This is called the standard streams in bash. There are three standard streams in bash:

- **Standard Input (stdin)** - This is the input stream that is used to read input from the user. By default, this is the keyboard.
- **Standard Output (stdout)** - This is the output stream that is used to display output to the user. By default, this is the terminal.
- **Standard Error (stderr)** - This is the error stream that is used to display error messages to the user. By default, this is the terminal.

The streams can also be redirected to/from files or other commands, as we have seen in Chapter **??**.

1.6.1 Reading Input

To read input from the user, we can use the `read` command. The `read` command reads a line of input from the stdin and stores it in a variable.

Optionally, we can also provide a prompt to the user, which is displayed before the user enters the input.

```
$ read -p "Enter your name: " name
Enter your name: Alice
$ echo "Hello, $name!"
Hello, Alice!
```

This reads the input from the user and stores it in the `name` variable, which is then used to display a greeting message.

We can use the same command inside a script to read input from the user, or take in any data passed in standard input.

```
$ cat input.sh
read line
echo "STDIN: $line"
$ echo "Hello, World!" | bash input.sh
STDIN: Hello, World!
```

However, the `read` command reads only one line of input. If we want to read multiple lines of input, we can use a loop to read input until the user enters a specific keyword.

```
$ cat multiline.sh
while read line; do
  echo "STDIN: $line"
done
$ cat file.txt
```

```
Hello World
This is a multiline file
We have three lines in it
$ cat file.txt | bash multiline.sh
STDIN: Hello World
STDIN: This is a multiline file
STDIN: We have three lines in it
```

Here we are reading input from the file `file.txt` and displaying each line of the file.

The redirection can be done more succinctly by using the input redirection operator <. The pipe was used to demonstrate that the input can be coming from the output of any arbitary commands, not just the content of a file.

```
$ bash multiline.sh < file.txt
STDIN: Hello World
STDIN: This is a multiline file
STDIN: We have three lines in it
```

1.7 Conditionals

Often times, in scripts, we require to decide between two blocks of code to execute, depending on some non-static condition. This can be either based on some value inputted by the user, a value of a file on the filesystem, or some other value fetched from the sensors or over the internet. To facilitate branching in the scripts, bash provides multiple keywords and commands.

1.7.1 Test command

The `test` command is a command that checks the value of an expression and either exits with a exit code of 0 [1] if the expression is true, or exits with a non-zero exit code if the expression is false.

1: Exit code 0 denotes success in POSIX

It has a lot of unary and binary operators that can be used to check various conditions.

Here we evaluate two equations, 1 = 1 and 5 > 7. The first equation is true, so the `test` command exits with a 0 exit code, while the second equation is false, so the `test` command exits with a 1 exit code.

```
$ test 1 -eq 1
$ echo $?
0
$ test 5 -gt 7
$ echo $?
1
```

String condtions

`test` can check for unary and binary conditions on strings.

- `-z` - True if the string is empty
- `-n` - True if the string is not empty
- `=` - True if the strings are equal
- `!=` - True if the strings are not equal

- `<` - True if the first string is lexicographically less than the second string
- `>` - True if the first string is lexicographically greater than the second string

Unary Operators

The `-z` and `-n` flags of `test` check if a string is empty or not.

```
$ var="apple"
$ test -n "$var" ; echo $?
0
$ test -z "$var" ; echo $?
1
```

```
$ var=""
$ test -n "$var" ; echo $?
1
$ test -z "$var" ; echo $?
0
```

Binary Operators

The `=`, `!=`, `<`, and `>` flags of `test` check if two strings are equal, not equal, less than, or greater than each other.

```
$ var="apple"
$ test "$var" = "apple" ; echo $?
0
$ test "$var" != "apple" ; echo $?
1
```

```
$ var="apple"
$ test "$var" = "banana" ; echo $?
1
$ test "$var" != "banana" ; echo $?
0
```

```
$ var="apple"
$ test "$var" \< "banana" ; echo $?
0
$ test "$var" \> "banana" ; echo $?
1
```

We are escaping the `>` and `<` characters as they have special meaning in the shell. If we do not escape them, the shell will try to redirect the input or output of the command to/from a file. We can also quote the symbols instead of escaping them.

Numeric conditions

`test` can also check for unary and binary conditions on numbers.

- `-eq` - True if the numbers are equal
- `-ne` - True if the numbers are not equal
- `-lt` - True if the first number is less than the second number
- `-le` - True if the first number is less than or equal to the second number
- `-gt` - True if the first number is greater than the second number
- `-ge` - True if the first number is greater than or equal to the second number

```
$ test 5 -eq 5 ; echo $?
0
$ test 5 -ne 5 ; echo $?
```

```
1
$ test 5 -ge 5 ; echo $?
0
$ test 5 -lt 7 ; echo $?
0
$ test 5 -le 7 ; echo $?
0
$ test 5 -gt 7 ; echo $?
1
$ test 5 -ge 7 ; echo $?
1
```

File conditions

The most number of checks in `test` are for files. We can check if a file exists, if it is a directory, if it is a regular file, if it is readable, writable, or executable, and many more.

File Types

- `-e` and `-a` - if the file exists
- `-f` - if the file is a regular file
- `-d` - if the file is a directory
- `-b` - if the file is a block device
- `-c` - if the file is a character device
- `-h` and `-L` - if the file is a symbolic link
- `-p` - if the file is a named pipe
- `-S` - if the file is a socket

File Permissions

- `-r` - if the file is readable
- `-w` - if the file is writable
- `-x` - if the file is executable
- `-u` - if the file has the setuid bit set
- `-g` - if the file has the setgid bit set
- `-k` - if the file has the sticky bit set
- `-O` - if the file is effectively owned by the user
- `-G` - if the file is effectively owned by the user's group

Binary Operators

- `-nt` - if the first file is newer than the second file
- `-ot` - if the first file is older than the second file
- `-ef` - if the first file is a hard link to the second file

Other conditions

- `-o` - Logical OR (When used in binary operation)
- `-a` - Logical AND
- `!` - Logical NOT
- `-v` - if the variable is set
- `-o` - if the `shopt` option is set (when used as unary operator)

The test built-in can be invoked in two ways, either by using the `test` word or by using the `[` built-in. The `[` is a synonym for the `test` command, and is used to make the code more readable. If we use the `[` command, the last argument must be `]`. This makes it look like a shell syntax rather than a command, but we should keep in mind that the test command (even the `[` shortcut) is a command built-in, and not a shell keyword, thus we need to use a space after the command.

```
$ [ 1 -eq 1 ] ; echo $?
0
$ var=apple
$ [ "$var" = "apple" ] ; echo $?
0
```

Remark 1.7.1 The `test` and `[` exist as both a executable stored in the filesystem and also as a shell built-in in bash. However the built-in takes preference when we simply type the name. Almost all operators are same in both, however the executable version cannot check if a variable is set using the `-v` option since executables dont have access to the shell variables.

1.7.2 Test Keyword

Although we already have the `test` executable and its shorthand `[` as built-in, bash also provides a keyword `[[` which is a more powerful version of the `test` command. This is present in bash [2] but not present in the POSIX standard.

2: And other popular shells like zsh

```
$ type -a test
test is a shell builtin
test is /usr/bin/test
$ type -a [
[ is a shell builtin
[ is /usr/bin/[
$ type -a [[
[[ is a shell keyword
```

This is an improvement over the `test` command, as it has more features and is more reliable.

- It does not require us to quote variables
- It supports logical operators like `&&` and `||`
- It supports regular expressions
- It supports globbing

Quoting and Empty Variables

```
$ var=""
$ [ $var = "apple" ] ; echo $?
-bash: [: =: unary operator expected
2
$ [ "$var" = "apple" ] ; echo $?
1
```

As we can see, the `test` command requires us to quote the variables, otherwise it will throw an error if the variable is empty. However, the `[[` keyword does not require us to quote the variables.

```
$ var=""
$ [[ $var = "apple" ]] ; echo $?
1
```

Logical Operators

The logical operators to combine multiple conditions are `&&` and `||`. In the `test` command, we can use the `-a` and `-o` operators, or we can use `&&` and `||` outside by combining multiple test commands.

```
$ var="apple"
$ [ "$var" = "apple" -a 1 -eq 1 ] ; echo $?
0
$ [ "$var" = "apple" ] && [ 1 -eq 1 ] ; echo $?
0
```

But in the `[[` keyword, we can use the `&&` and `||` operators directly.

```
$ var="apple"
$ [[ "$var" = "apple" && 1 -eq 1 ]] ; echo $?
0
```

Comparison Operators

For string comparison, we had to escape the `>` and `<` characters in the `test` command, but in the `[[` keyword, we can use them directly.

```
$ var="apple"
$ [[ "$var" < "banana" ]] ; echo $?
0
```

Regular Expressions

The `[[` keyword also supports regular expressions using the `=~` operator.

If we want to check whether the variable `response` is `y` or `yes`, in `test` we would do the following.

```
$ response="yes"
$ [ "$response" = "y" -o "$response" = "yes" ] ; echo $?
0
```

But in `[[` we can use the `=~` operator to check if the variable matches a regular expression.

```
$ response="yes"
$ [[ "$response" =~ ^y(es)?$ ]] ; echo $?
0
```

It also uses **ERE** (Extended Regular Expressions) by default, so we can use the `?` operator without escaping it to match 0 or 1 occurrence of the previous character.

Warning 1.7.1 When using regular expressions in bash scripts, we should never quote the regular expression, as it will be treated as a string and not a regular expression.

Globbing

We can also simply match for any response starting with `y` using globbing.

```
$ response="yes"
$ [[ "$response" == y* ]] ; echo $?
0
```

Double Equals

In the `[[` keyword, we can use the `==` operator to check for string equality, which is a widely known construct from most languages.

1.8 If-elif-else

1.8.1 If

The `if` statement is used to execute a block of code if a condition is true. If the condition is false, the block of code is skipped.

The end of the `if` statement is denoted by the `fi` keyword, which is the reverse of the `if` keyword.

The syntax of the `if` statement is as follows:

```
if command ; then
  # code to execute if the command is successful
else
  # code to execute if the command is not successful
fi
```

Unlike most programming language that takes a boolean expression, the `if` statement in bash takes a command, and executes the command. If the command exits with a 0 exit code, the block of code after the `then` keyword is executed, otherwise the block of code after the `else` keyword is executed.

This allows us to use any command that we have seen so far in the `if` statement. The most used command is the `test` command, which is used to check conditions.

```
$ var="apple"
$ if [ "$var" =~ ^a.* ] ; then
>   echo "The variable starts with a"
> fi
The variable starts with a
```

However, we can also use other commands, like the `grep` command, to check if a file contains a pattern.

```
$ cat file.txt
Hello how are you
$ if grep -q "how" file.txt ; then
>   echo "The file contains the word how"
> fi
The file contains the word how
$ if grep -q "world" file.txt ; then
>   echo "The file contains the word world"
> fi
```

The first `if` statement checks if the file contains the word `how`, which is present, and thus prints the message to the screen. The second `if` statement checks if the file contains the word `world`, which is not present, and thus does not print anything.

1.8.2 Else

The `else` keyword is used to execute a block of code if the command in the `if` statement is not successful.

```
$ var="apple"
$ if [ "$var" = "banana" ] ; then
>   echo "The variable is banana"
> else
>   echo "The variable is not banana"
> fi
The variable is not banana
```

The combination of the `if` and `else` keywords allows us to execute different blocks of code depending on the condition; this is one of the most used construct in scripting.

```
$ read -p "Enter a number: " num
Enter a number: 5
$ if [[ $num -gt 0 ]] ; then
>   echo "The number is positive"
> else
>   echo "The number is negative"
> fi
The number is positive
```

However, notice that this allows us only two branches, one for the condition being true, and one for the condition being false. Due to this, we have misclassified the number 0 as positive, which is not correct.

To fix this, we can use the `elif` keyword, which is short for `else if`.

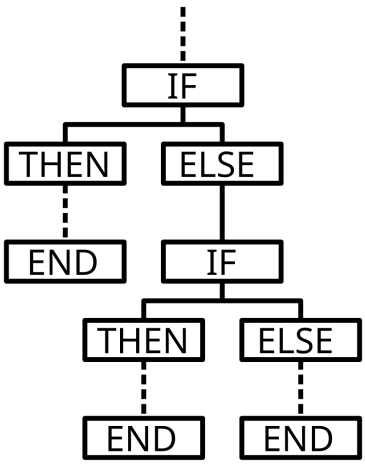

Figure 1.1: Flowchart of the `if`, `elif`, and `else` construct

1.8.3 Elif

If the command in the `if` statement is not successful, we can run another command using the `elif` keyword and decide the branching based on its exit status.

```
$ read -p "Enter a number: " num
Enter a number: 0
$ if [[ $num -gt 0 ]] ; then
>   echo "The number is positive"
> elif [[ $num -lt 0 ]] ; then
>   echo "The number is negative"
> else
>   echo "The number is zero"
> fi
The number is zero
```

Here we have added an `elif` statement to check if the number is less than 0, and if it is, we print that the number is negative, finally if both

the commands are unsuccessful, we execute the statements in the `else` block.

1.9 Exit code inversion

We can also invert the condition using the `!` operator.

```
$ var=apple
$ if ! [ "$var" = "banana" ] ; then
>   echo "The variable is not banana"
> fi
The variable is not banana
```

1.10 Mathematical Expressions as if command

We can also use the `((` keyword to evaluate mathematical expressions in the `if` statement. This environment does not print any output to the standard output, rather, it exits with a zero exit code if the result of the mathematical evaluation is non-zero, and exits with a non-zero exit code if the evaluation results to zero. This is useful as 0 is false in mathematics but 0 is success exit code in the shell.

```
$ if (( 5 > 3 )) ; then
>   echo "5 is greater than 3"
> fi
5 is greater than 3
```

However, this environment only supports integers, and not floating point numbers.

1.11 Command Substitution in if

We can also run a command in the `if` statement, and compare the output of the command to a value using the `test` command and the `$(())` construct.

```
$ if [ $(wc -l < file.txt) -gt 100 ] ; then
>   echo "The file is big"
> fi
The file is big
```

1.12 Switch

Definition 1.12.1 (Switch Case) A switch case is a syntactical sugar that helps match a variable's value against a list of options (or patterns) and executes the block of code corresponding to the match.

The branching achieved by **switch case** can also be achieved by using multiple `if-elif-else` statements, but the switch case is more readable and concise.

Syntax

The `case` block is ended with the keyword `esac`, which is the reverse of the `case` keyword.

```
case var in
  pattern1)
    # code to execute if var matches pattern1
    ;;
  pattern2)
    # code to execute if var matches pattern2
    ;;
  *)
    # code to execute if var does not match any pattern
    ;;
esac
```

The `case` keyword is followed by the variable to match against, and the `in` keyword. Then we have multiple patterns, each followed by a block of code to execute if the variable matches the pattern.

The `;;` keyword is used to denote the end of a block of code, and the start of the next pattern. This ensures that the script does not continue running the statements of the other blocks after it as well.

The `*` pattern is a wildcard pattern, and matches any value that does not match any of the other patterns.

The patterns in the switch case are similar to globs rather than regular expressions, so we cannot use regular expressions in the switch case.

Although sometimes we actually want to execute the blocks after the match. This is called fall through.

1.12.1 Fall Through

In this example, the script will print both the statements "Multiple of 10" and "Multiple of 5" for numbers ending with zero because of the fall through `;&` instead of `;;`.

If we replace the `;;` with `;&` or `;;&` then the control will flow into the next block.

```
$ cat switch.sh
read -p "Enter number: " num
case "$num" in
  *0) echo "Multiple of 10" ;&
  *5) echo "Multiple of 5" ;;
  *) echo "Not Multiple of 5" ;;
esac
$ bash switch.sh
Enter number: 60
Multiple of 10
Multiple of 5
```

1.12.2 Multiple Patterns

We can also have multiple patterns for a single block of code by separating the patterns with the `|` operator.

```
$ cat switch.sh
read -p "Enter digit: " num
case "$num" in
  1|2|3) echo "Small number" ;;
  4|5|6) echo "Medium number" ;;
  7|8|9) echo "Large number" ;;
  *) echo "Invalid number" ;;
esac
$ bash switch.sh
Enter digit: 5
Medium number
```

1.13 Select Loop

Definition 1.13.1 (Select Loop) A select loop is a construct that is used to create a menu in the shell script. It takes a list of items and displays them to the user, and then waits for the user to select an item. The selected item is stored in a variable, and the block of code corresponding to the selected item is executed. This repeats infinitely, until the user stops it.

```
$ cat select.sh
select choice in dog cat bird stop
do
  case $choice in
    dog) echo "A dog barks" ;;
    cat) echo "A cat meows" ;;
    bird) echo "A bird chirps" ;;
    stop) break ;;
    *) echo "Invalid choice" ;;
  esac
done
$ bash select.sh
1) dog
2) cat
3) bird
4) stop
#? 1
A dog barks
#? 2
A cat meows
#? 3
A bird chirps
#? 4
```

A select loop is simply a syntactical sugar for a while loop that displays a menu to the user and waits for the user to select an option and reads it using `read`. The conditional branching is done using a `case` statement.

The menu output is actually displayed on the standard error stream, so that it is easy to split the conditional output from the menu output.

```
$ bash select.sh > output.txt
1) dog
2) cat
3) bird
```

```
4) stop
#? 1
#? 2
#? 3
#? 4
$ cat output.txt
A dog barks
A cat meows
A bird chirps
```

To stop the select loop, we can use the `break` keyword. We will see more about loops in the next section.

1.14 Loops

If we want to execute a block of code multiple times, we can use loops. There are three types of loops in bash:

- **For loop** - Used to iterate over a list of items or a fixed number of times
- **While loop** - Used to execute a block of code as long as a condition is true
- **Until loop** - Used to execute a block of code as long as a condition is false

1.14.1 For loop

In bash, we can use the `for` loop to iterate over a list of items, called a **for-each** loop [3], or a range of numbers [4].

3: This is similar to how the for loop in Python works.

4: This is similar to how the for loop in C and C like languages works.

For-each loop

```
$ cat foreach.sh
for item in mango banana strawberry; do
  echo "$item shake"
done
$ bash foreach.sh
mango shake
banana shake
strawberry shake
```

As we saw in Chapter **??**, there are three ways to iterate over an array in bash. We can either treat the entire array as a single element (`"${arr[*]}"`), we can break the elements by spaces (`${arr[@]}`), or we can split as per the array elements, preserving multi-word elements (`"${arr[@]}"`).

Treating entire array as a single element:

```
$ cat forsplit.sh
name=("Sayan" "Alice" "John Doe")
for name in "${name[*]}"; do
  echo "Name: $name"
done
$ bash forsplit.sh
Name: Sayan Alice John Doe
```

Splitting by spaces:

```
$ cat forsplit.sh
name=("Sayan" "Alice" "John Doe")
for name in ${name[@]}; do
  echo "Name: $name"
done
$ bash forsplit.sh
Name: Sayan
Name: Alice
Name: John
Name: Doe
```

Preserving multi-word elements:

```
$ cat forsplit.sh
name=("Sayan" "Alice" "John Doe")
for name in "${name[@]}"; do
  echo "Name: $name"
done
$ bash forsplit.sh
Name: Sayan
Name: Alice
Name: John Doe
```

We can also dynamically generate a list of numbers in a range using the `{start..end}` syntax or the `seq` command.

```
$ cat range.sh
for i in {1..5}; do
  echo "Number: $i"
done
$ bash range.sh
1
2
3
4
5
```

```
$ cat range.sh
for i in $(seq 1 5); do
  echo "Number: $i"
done
$ bash range.sh
1
2
3
4
5
```

Differences between range expansion and seq command

Although both `seq` and the **range expansion** bash syntax have similar functionality, they have slightly different behaviour, as seen in Table 1.1.

Letters:

```
$ echo {a..e}
a b c d e
$ seq a e
seq: invalid floating point argument: 'a'
Try 'seq --help' for more information.
```

Table 1.1: Differences between range expansion and seq command

Range Expansion	Seq Command
It is a bash feature hence it is faster	It is an external command
Only integer step size	Fractional step size is allowed
Works on letters	Works only on numbers
Step size should be the third argument	Step size should be the second argument
Output is space separated	Output is newline separated
Start range cannot be omitted	Start range is 1 by default

Position of step size:

```
$ echo {1..10..2}
1 3 5 7 9
$ seq 1 2 10
1
3
5
7
9
```

Fractional step size:

Note that if the shell is not able to expand a range expansion due to invalid syntax it will not throw an error, rather it will simply keep it unexpanded. This is similar to how path expansion works.

```
$ seq 1 0.5 2
1.0
1.5
2.0
$ echo {1..2..0.5}
{1..2..0.5}
```

Default start range:

```
$ seq 5
1
2
3
4
5
```

Different between python's range and seq command:

The `seq` command is similar to the `range` function in Python, but there are some differences between the two.

Table 1.2: Differences between seq and python's range function

Python's Range	Seq Command
Start of range is 0 by default	Start range is 1 by default
Order of parameters is start,end,step	Order of parameters is start,step,end
End range is exclusive	End range is inclusive

1.14.2 C style for loop

Bash also supports C-style for-loops, where we declare a variable, a condition for the loop, and an increment command.

```
$ cat cstyle.sh
for ((i=0; i<5; i++)); do
  echo "Number: $i"
done
$ bash cstyle.sh
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
```

We can also have multiple variables in the C-style for loop.

```
$ cat cstyle.sh
begin1=1
begin2=10
finish=10
for (( a=$begin1, b=$begin2; a < $finish; a++, b-- )); do
  echo $a $b
done
$ bash cstyle.sh
1 10
2 9
3 8
4 7
5 6
6 5
7 4
8 3
9 2
```

1.14.3 IFS

By default the for loop splits the input by tabs, spaces, and newlines. We can change the delimiter by changing the `IFS` variable.

Definition 1.14.1 (IFS) The `IFS` variable is the Internal Field Separator, and is used to split the input into fields.

It is used by for loop and other word splitting operations in bash.

Default value of `IFS` is `$' \t\n'`. If set to multiple characters, it will split the input by any of the characters. [5]

5: The `$''` syntax is used to denote ANSI escape sequences in the string in bash.
In this example we are reading the `$PATH` variable, which is a colon separated list of directories. We are changing the `IFS` variable to a colon, so that the for loop splits the input by colon.

```
$ cat ifs.sh
IFS=:
for i in $PATH; do
  echo $i
done
$ bash ifs.sh
/usr/local/sbin
/usr/local/bin
/usr/bin
```

We should remember to reset the `IFS` variable after using it, as it can cause unexpected behaviour in other parts of the script.

```
$ cat unsetifs.sh
IFS=:
# some code

var="a b c:d"
for i in $var; do
  echo $i
done
$ bash unsetifs.sh
a b c
d
```

Although we wanted to iterate over the elements by spliting by space, we ended up splitting by colon because we forgot to reset the `IFS` variable.

To reset the `IFS` variable, we can simply unset it.

```
$ cat unsetifs.sh
IFS=:
# some code

unset IFS
var="a b c:d"
for i in $var; do
  echo $i
done
$ bash unsetifs.sh
a
b
c:d
```

Unsetting the `IFS` variable will reset it to the default value of `$' \t\n'`.

However, setting and resetting the `IFS` variable can be cumbersome, so we can use a subshell to change the `IFS` variable only for the for loop.

```
$ cat subshellifs.sh
var="a b c:d"
(
IFS=:
for i in $var; do
  echo $i
done
)

for i in $var; do
  echo $i
done
$ bash subshellifs.sh
a b c
d
a
b
c:d
```

1.14.4 While loop

The `while` loop is used to execute a block of code as long as a condition is true. This is useful if we do not know the number of iterations beforehand,

rather we know the condition that should be satisfied.

Just like the `if` statement, the `while` loop also takes a command, and executes the block of code if the command is successful. This means we can run any arbitrary command in the `while` loop condition checking.

```
$ cat while.sh
i=5
while [ $i -gt 0 ]; do
  echo  "i is $i"
  ((i--))
done
$ bash while.sh
i is 5
i is 4
i is 3
i is 2
i is 1
```

Here we are using the `test` command to check if the variable `i` is greater than 0, and if it is, we print the value of `i` and decrement it by 1.

However, `test` is not the only command that we can use in the `while` loop condition.

Here we are using the `grep` command to check if the password contains any special character or not, and looping as long as the user enters a strong password.

Ideally when reading passwords, we should use the `read -s` command to hide the input from the user, however it is ommited from the example so that the input can be seen by the reader.

```
$ cat pass.sh
read -p "Enter password: " pass

while ! grep -q '[[:punct:]]' <<< "$pass" ; do
  echo "Password must contain at least one special character"
  read -p "Enter password: " pass
done
echo "Password is set"
$ bash pass.sh
Enter password: 123
Password must contain at least one special character
Enter password: abc
Password must contain at least one special character
Enter password: sayan
Password must contain at least one special character
Enter password: $ayan
Password is set
```

1.14.5 Until loop

The `until` loop is used to execute a block of code as long as a condition is false. This is simply a negation of the `while` loop. It is a syntactical sugar. The last example we saw in the `while` loop can be rewritten using the `until` loop to omit the `!` symbol.

```
$ cat pass.sh
read -p "Enter password: " pass

until grep -q '[[:punct:]]' <<< "$pass" ; do
  echo "Password must contain at least one special character"
  read -p "Enter password: " pass
done
echo "Password is set"
$ bash pass.sh
```

```
Enter password: 123
Password must contain at least one special character
Enter password: abc
Password must contain at least one special character
Enter password: sayan
Password must contain at least one special character
Enter password: $ayan
Password is set
```

1.14.6 Read in while

The `read` command is used to read one line of input from the user, however if we want to read multiple lines of input, and we do not know the number of lines beforehand, we can use the `while` loop to read input until the user enters a specific value, or the input stops. [6]

6: The end of input is denoted by the `EOF` character, if we are reading input from standard input, we can press `Ctrl+D` to send the `EOF` character and mark the end of input.

```
$ cat read.sh
while read line; do
  echo "Line is $line"
done
$ bash read.sh
hello
Line is hello
this is typed input
Line is this is typed input
now press ctrl+D
Line is now press ctrl+D
```

We can also read from a file using the < operator. Since the entire while loop is a command, we can use the < operator to redirect the input to the while loop after the ending `done` keyword.

Here we are reading the `/etc/passwd` file line by line and printing the username. The username is the first field in the `/etc/passwd` file, which is separated by a colon. The username is extracted by removing everything after the first colon using shell variable manipulation.

```
$ cat read.sh
while read line; do
  echo ${line%%:*}
done < /etc/passwd
$ bash read.sh
root
bin
daemon
mail
ftp
http
nobody
dbus
systemd-coredump
systemd-network
```

We can also use the `IFS` variable to split each row by a colon, and extract the username.

In this example, we are setting the IFS only for the while loop, and it gets reset after the loop. If we provide multiple variables to the `read` command, it will split the input by the `IFS` variable and assign the split values to the variables.

```
$ cat read.sh
while IFS=: read username pass uid gid gecos home shell; do
  echo $username - ${gecos:-$username}
done < /etc/passwd
$ bash read.sh
root - root
bin - bin
```

```
daemon - daemon
mail - mail
ftp - ftp
http - http
nobody - Kernel Overflow User
dbus - System Message Bus
systemd-coredump - systemd Core Dumper
systemd-network - systemd Network Management
```

If the number of variables provided to the `read` command is less than the number of fields in the input, the remaining fields are stored in the last variable. This can be utilized to read only the first field of the input, and discard the rest.

In this example, we are reading only the first field of the input, and discarding the rest. In bash (and many other languages) the underscore variable is used to denote a variable that is not to be used.

```
$ cat read.sh
while IFS=: read username _; do
  echo $username
done < /etc/passwd
$ bash read.sh
root
bin
daemon
mail
ftp
http
nobody
dbus
systemd-coredump
systemd-network
```

1.14.7 Break and Continue

The `break` and `continue` keywords are used to control the flow of the loop.

- **Break** - The `break` keyword is used to exit the loop immediately. This skips the current iteration, and also all the next iterations.
- **Continue** - The `continue` keyword is used to skip the rest of the code in the loop and go to the next iteration.

Break:

```
$ cat pat.sh
for i in {1..5}; do
  for j in {1..5}; do
    if [[ "$j" -eq 3 ]]; then
      break;
    fi
    echo -n $j
  done
  echo
done
$ bash pat.sh
12
12
12
```

```
12
12
```

Continue:

```
$ cat pat.sh
for i in {1..5}; do
  for j in {1..5}; do
    if [[ "$j" -eq 3 ]]; then
      continue;
    fi
    echo -n $j
  done
  echo
done
$ bash pat.sh
1245
1245
1245
1245
1245
```

Unlike most languages, the `break` and `continue` keywords in bash also takes an optional argument. This argument is the number of loops to break or continue. If we change the `break` keyword to `break 2`, it will break out of the outer loop, and not just the inner loop.

```
$ cat pat.sh
for i in {1..5}; do
  for j in {1..5}; do
    if [[ "$j" -eq 3 ]]; then
      break 2;
    fi
    echo -n $j
  done
  echo
done
$ bash pat.sh
12
```

The same works for the `continue` keyword as well.

```
$ cat pat.sh
for i in {1..5}; do
  for j in {1..5}; do
    if [[ "$j" -eq 3 ]]; then
      continue 2;
    fi
    echo -n $j
  done
  echo
done
$ bash pat.sh
1212121212
```

1.15 Functions

If the script is too long, it is usually better to split the script into multiple concise functions that does only one task.

There are three ways to define a function in bash:

Without the function keyword:

```
abc(){
  commands
}
```

function keyword with brackets

```
function abc(){
  commands
}
```

function keyword without brackets

```
function abc {
  commands
}
```

Example:

```
$ cat functions.sh
sayhi(){
    echo "Hello, $1"
}

sayhi "John"
$ bash functions.sh
Hello, John
```

Arguments in Functions

Just like we can use `$1` inside a script to refer to the first argument, we can use `$1` inside a function to refer to the first argument passed to the function.

The arguments passed to the script are not available inside the function, instead the arguments passed to the function are available inside the function.

```
$ cat fun.sh
fun(){
  echo $1 $2
}
echo "Full name:" $1 $2
fun "Hello" "$1"
$ bash fun.sh John Appleseed
Full name: John Appleseed
Hello John
```

Return value of a function

Functions in bash do not return a value, rather they exit with a status code. However, functions can print the value to the standard output, and the caller can capture the output of the function using command substitution.

In this example, we are creating a simple calculator script that takes two operands and an operator, and returns the result. The operator selection is done using a select loop, and the operands are read using `read`. We have modularized the script by creating two functions, `add` and `mul`, that takes two operands and returns the result.

```
$ cat calc.sh
add(){
  echo $(($1 + $2))
}

mul(){
  echo $(($1 * $2))
}

select operator in plus multiply ; do
  read -p "Operand 1: " op1
  read -p "Operand 2: " op2
  case "$operator" in
    plus) answer=$(add $op1 $op2) ;;
    multiply) answer=$(mul $op1 $op2) ;;
    *) echo "Invalid option" ;;
  esac
  echo "Answer is $answer"
done
$ bash calc.sh
1) plus
2) multiply
#? 1
Operand 1: 5
Operand 2: 4
Answer is 9
#? 2
Operand 1: 6
Operand 2: 3
Answer is 18
#?
```

However, sometimes we may want to return from a function as a means to exit the function early. We may also want to signal the success or failure of the function. Both of these can be done using the `return` shell built-in.

In this example, we have created a function that prints the numbers from 0 to the number passed to the function, but if the number passed is negative, the function returns early with a status code of 1. If the number if greater than 9, it only prints till 9, and then returns.

```
$ cat return.sh
fun(){
  if [[ $1 -lt 0 ]]; then
    return 1
  fi
  local i=0
  while [[ $i -lt $1 ]]; do
    if [[ $i -gt 9 ]]; then
      echo
      return
    fi
    echo -n $i
    ((i++))
  done
  echo
```

```
}

fun 5
echo return value: $?
fun 15
echo return value: $?
fun -5
echo return value: $?
$ bash return.sh
01234
return value: 0
0123456789
return value: 0
return value: 1
```

Remark 1.15.1 If no value is provided to the `return` command, it returns the exit status of the last command executed in the function.

Local variables in functions

By default, all variables in bash are global, and are available throughout the script. Even variables defined inside a function are global, and are available outside the function. This might cause confusion, as the variable might be modified by the function, and the modification might affect the rest of the script.

To make a variable local to a function, we can use the `local` keyword.

```
$ cat local.sh
fun(){
  a=5
  local b=10
}

fun
echo "A is $a"
echo "B is $b"
$ bash local.sh
A is 5
B is
```

If a variable is declared using the `declare` keyword, it is global if defined outside a function and local if defined inside a function.

As it is seen in the example, the variable `a` is available outside the function, but the variable `b` is not available outside the function since it is defined using the `local` shell built-in.

1.16 Debugging

Sometimes, especially when writing and debugging scripts, we may want to print out each line that the interpreter is executing, so that we can trace the control flow and find out any logical errors.

This can be done by setting the `x` flag of `bash` using `set -x`.

```
$ cat debug.sh
fun(){
  echo $(($1 + $2))
}

fun $(fun $(fun 1 2) 3) 4
$ bash debug.sh
10
```

In the above script, we are calling the `fun` function multiple times, and it is difficult to trace the control flow.

To trace the control flow, we can set the `x` flag using the `set` command.

```
$ cat debug.sh
set -x
fun(){
  echo $(($1 + $2))
}

fun $(fun $(fun 1 2) 3) 4
$ bash debug.sh
+++ fun 1 2
+++ echo 3
++ fun 3 3
++ echo 6
+ fun 6 4
+ echo 10
10
```

Now we can see the control flow of the script, and we can trace the execution of the script. The `+` is the `PS4` prompt, which denotes that the line printed is a trace line and not real output of the script.

Remark 1.16.1 The trace output of a script is printed to the standard error stream, so that it does not interfere with the standard output of the script.

The `PS4` prompt is repeated for each level of the function call, and is incremented by one for each level. This helps us visualize the call stack and order of execution of the script.

1.17 Recursion

Just like other programming languages, bash also supports recursion. However, since functions in bash do not return a value, it becomes terse to use recursion in bash by using command substitutions.

```
fibo(){
  if [[ $1 -le 2 ]]; then
    echo 1
  else
    echo $(($(fibo $(($1-1))) + $(fibo $(($1-2)))))
  fi
}

```

```
for i in {1..10}; do
  fibo $i
done
```

In the above example, we are calculating the first ten elements of the fibonacci series using recursion. We need to use **command substitution** to capture the output of the function, and then use **mathematical evaluation** environment to add the two values.

However, most recursive solutions, including this one, are not efficient, and are not recommended for use in bash as they will be very slow. If we time the function with a argument of 20, we see it takes a lot of time.

```
$ time fibo 20
6765

real    0m13.640s
user    0m10.010s
sys     0m3.187s
```

An iterative solution would be much faster and efficient.

```
$ cat fibo.sh
fibo(){
  declare -i a=0
  declare -i b=1
  for (( i=1; i<=$1; i++ )); do
    declare -i c=$a+$b
    a=$b
    b=$c
  done
  echo $a
}

time fibo 40
$ bash fibo.sh
102334155

real    0m0.001s
user    0m0.001s
sys 0m0.000s
```

The iterative solution is much faster and efficient than the recursive solution.

1.18 Shell Arithmetic

We have already seen the mathematical evaluation environment in bash, which is used to evaluate mathematical expressions.

```
$ echo $((1 + 2))
3
```

However, the mathematical evaluation environment is limited to integer arithmetic, and does not support floating point arithmetic.

1.18.1 bc

A more powerful way to do arithmetic in bash is to use the `bc` command.

Definition 1.18.1 (BC) `bc` is an arbitrary precision calculator language, and is used to do floating point arithmetic in bash.

```
$ bc <<< "1.5 + 2.5"
4.0
```

However, `bc` by default will set the scale to 0, and will truncate the result to an integer.

```
$ bc <<< "10/3"
3
```

This can be changed by setting the `scale` variable in `bc`.

```
$ bc <<< "scale=3; 10/3"
3.333
```

We can also use the `-l` flag to load the math library in `bc`, which provides more mathematical functions and also sets the scale to 20.

```
$ bc -l <<< "10/3"
3.33333333333333333333
```

`bc` can also be started in an interactive mode, which is a REPL. [7]

7: **REPL - Read, Evaluate, Print, Loop** is an interactive interpreter of a language.

`bc` is not just a calculator, rather it is a full fledged programming language, and can be used to write scripts.

`bc` supports all the basic arithmetic operations, and also supports the `if` statement, `for` loop, and `while` loop and other programming constructs. It is similar to C in syntax, and is a powerful language.

```
$ cat factorial.bc
define factorial(n) {
  if(n==1){
    return 1;
  }
  return factorial(n-1) * n;
}
$ bc factorial.bc <<< "factorial(5)"
120
```

Read the man page of `bc` to know more about the language.

```
$ man bc
```

1.18.2 expr

Definition 1.18.2 (EXPR) `expr` is a command line utility that is used to evaluate expressions.

As `expr` is a command line utility, it is not able to access the shell variables directly, rather we need to use the `$` symbol to expand the variable with the value before passing the arguments to `expr`.

```
$ expr 1 + 2
3
```

The spaces around the operator are necessary, as `expr` is a command line utility, and the shell will split the input by spaces.

The operand needs to be escaped or quoted if it has a special meaning in the shell.

```
$ expr 10 \* 3
30
$ expr 10 '*' 3
30
$ ls
$ expr 10 * 3
30
$ touch '+'
$ expr 10 * 3
13
```

The * symbol is a special character in the shell, and is used for path expansion. If the folder is empty, then it remains as *, but if the directory has files, then * will expand to the sorted list of all the files. In this example we show this by creating a file with the name as +, thus 10*3 actually expands to 10+3 and gives the output of 13.

Like the mathematical environment, `expr` exits a zero exit code if the expression evaluates to a non-zero value, and exits with a non-zero exit code if the expression evaluates to zero.

This inversion is useful in `if` and `while` loops.

```
$ expr 5 '>' 6
0
$ echo $?
1
$ expr 5 '<' 6
1
$ echo $?
0
```

`expr` can also be used to match regex patterns. Unlike most other commands, the regex pattern is always anchored to the start of the string.

```
$ expr hello : h
1
$ expr hello : e
0
$ expr hello : .*e
2
$ expr hello : .*l
4
```

As seen, the regex pattern is always anchored to the start of the string. The matching is done greedily, and the maximum possible match is taken. `expr` prints the length of the match, and not the match itself.

However, if we want to actually print the match instead of the length, we can enclose the regex pattern in escaped parentheses.

```
$ expr hello : '\(.*l\)'
hell
```

Other string operations that `expr` supports are:

- `length` - Returns the length of the string.
- `index` - Returns the index of the first occurrence of the substring.
- `substr` - Returns the substring of the string.

```
$ expr length hello
5
```

```
$ expr index hello e
2
```

```
$ expr substr hello 2 3
ell
```

The index is 1 based in `expr`, and not 0 based. For the `substr` command, the first argument is the string, the second argument is the starting index, and the third argument is the length of the substring.

1.19 Running arbitary commands using source, eval and exec

Just like we can use the `source` command to run a script file in the current shell itself, we can also run any arbitary command directly using `eval` without needing a file.

```
$ eval date
Wed Jul 31 11:00:42 PM IST 2024
$ cat script.sh
echo "Hello"
$ source script.sh
Hello
$ eval ./script.sh
Hello
```

As it can run any command, it can also run a script file. But the `source` command cannot run a command without a file. However, this can be circumvented by using the `/dev/stdin` file.

```
$ source /dev/stdin <<< "echo Hello"
Hello
```

Warning 1.19.1 We should be careful when using the `eval` command, as it can run any arbitary command in the current shell, and can be a security risk.

1.19.1 exec

Similar to the `eval` command, the `exec` command can also be used to run arbitary commands in the same environment without creating a new environment. However, the shell gets replaced by the command being run, and when the command exits, the terminal closes. If the command fails to run then the shell is preserved.

Exercise 1.19.1 Open a new terminal and run the command `exec sleep 2`. Observe that the terminal closes after 2 seconds.

1.20 Getopts

Getopts is a built-in command in bash that is used to parse command line arguments. It is a syntactical sugar that helps us parse command line arguments easily.

The first argument to the `getopts` command is the string that contains the options that the script accepts. The second argument is the name of the variable that will store the option that is parsed. . . `getopts` reads each argument passed to the script one by one, and stores the option in the variable, and the argument to the option in the `$OPTARG` variable. It needs to be executed as many times as the number of options passed to the script. As this number is often unknown, it is usually executed in a `while` loop, since the `getopts` command returns a non-zero exit code once all the passed options are checked.

```
$ cat optarg.sh
while getopts ":a:bc:" flag; do
    echo "flag -$flag, Argument $OPTARG";
done
$ bash optarg.sh -a 1 -b -c 2
flag -a, Argument 1
flag -b, Argument
flag -c, Argument 2
```

The colon after the option denotes that the option requires an additional argument. The colon at the start of the string denotes that the script should not print an error message if an invalid option is passed.

Without the leading colon:

```
$ cat optarg.sh
while getopts "a:" flag; do
    echo "flag -$flag, Argument $OPTARG";
done
$ bash optarg.sh -a 1 -b
flag -a, Argument 1
optarg: illegal option -- b
flag -?, Argument
$ bash optarg.sh -a
optarg: option requires an argument -- a
flag -?, Argument
```

With the leading colon:

```
$ cat optarg.sh
while getopts ":a:" flag; do
    echo "flag -$flag, Argument $OPTARG";
done
$ bash optarg.sh -a 1 -b
flag -a, Argument 1
flag -?, Argument b
$ bash optarg.sh -a
flag -:, Argument a
```

If the option is not passed, the `$OPTARG` variable is set to the option itself, and the `flag` variable is set to `:`.

If an illegal option is passed, the `flag` variable is set to `?`, and the `$OPTARG` variable is set to the illegal option.

These let the user print a custom error message if an illegal option is passed or if an option that requires an argument is passed without an argument.

1.20.1 With case statement

Usually the `getopts` command is used with a `case` statement to execute the code for each option.

```
$ cat optarg.sh
time="Day"
while getopts ":n:mae" opt; do
  case $opt in
    n) name=$OPTARG ;;
    m) time="Morning" ;;
    a) time="Afternoon" ;;
    e) time="Evening" ;;
    \?) echo "Invalid option: $OPTARG" >&2 ;;
  esac
done
echo -n "Good $time"
if [ -n "$name" ]; then
  echo ", $name!"
else
  echo "!"
fi
$ bash optarg
Good Day!
$ bash optarg -a
Good Afternoon!
$ bash optarg -e
Good Evening!
$ bash optarg -m
Good Morning!
$ bash optarg -an Sayan
Good Afternoon, Sayan!
$ bash optarg -mn Sayan
Good Morning, Sayan!
$ bash optarg -en Sayan
Good Evening, Sayan!
$ bash optarg -n Sayan
Good Day, Sayan!
$ bash optarg -n Sayan -a
Good Afternoon, Sayan!
```

The error printing can also be supressed by setting the `OPTERR` shell variable to 0.

1.21 Profile and RC files

There are two kinds of bash environments:

- **Login shell** - A login shell is a shell that is started after a user logs in. It is used to set up the environment for the user.
- **Non-login shell** - A non-login shell is a shell that is started after the user logs in, and is used to run commands.

When a non-login shell is started, it reads the `~/.bashrc` file, and the `/etc/bash.bashrc` file. [8]

8: The **rc** in **bashrc** stands for **run command**. This is a common naming convention in Unix-like systems, where the configuration files are named with the extension **rc**.

When a login shell is started, along with the **run command** files, it reads the `/etc/profile` file, and then reads the `~/.bash_profile` and `~/.profile` file.

We make most of the configuration changes in the `~/.bashrc` file, as it is read by both login and non-login shells, and is the most common configuration file. Make sure to backup the `~/.bashrc` file before making any changes, as a misconfiguration can cause the shell to not start.

1.22 Summary

In this chapter, we learned about the control flow constructs in bash, and how to use them to author scripts to automate your work.

Table 1.3: Summary of the bash constructs

Construct	Description
`if`	Used to execute a block of code based on a condition.
`case`	Used to execute a block of code based on a pattern.
`for`	Used to iterate over a list of elements.
`while`	Used to execute a block of code as long as a condition is true.
`until`	Used to execute a block of code as long as a condition is false.
`break`	Used to exit the loop immediately.
`continue`	Used to skip the rest of the code in the loop and go to the next iteration.
`read`	Used to read input from the user.
`unset`	Used to unset a variable.
`local`	Used to declare a variable as local to a function.
`return`	Used to return from a function.
`source`	Used to run a script file in the current shell.
`eval`	Used to run arbritary commands in the current shell.
`exec`	Used to run arbritary commands in the current shell.
`getopts`	Used to parse command line arguments.