CSCI 2132: Software Development

Lab 4/5: Shell Scripting

Synopsis

In this lab, you will:

- Learn to work with command-line arguments in shell scripts
- Learn to capture command output in variables
- Learn to work with exit codes of programs
- Learn the basics of using `sed`
- Use shell scripts to automate some simple data analytics tasks
- Learn how to parse command-line arguments in shell scripts

Contents

Overview

This lab is split into two parts. In Part 1, you will familiarize yourself with program exit codes, both how to use them in your shell and how to make shell scripts generate informative exit codes. You will learn how to use the `sed` command to apply simple transformations to the lines of a file. (`sed` allows you to write fairly complex programs to manipulate text, but we will not explore this in depth here.) Armed with these tools, you will write a simple script in Part 2 that allows you to perform some simple analysis tasks on a CSV file representing a simple employee database. (The analysis task is simple. The scripts you develop to carry out this task are less simple.)

Note: I do not expect you to complete this lab in one session. Therefore, this is labelled as Lab 4/5; you will have time to finish this lab next week.

Step 1: Checking exit codes

Any program you run from the shell returns an exit code to the shell when it terminates. You can think of the program as a function you call from your shell and of the exit code as the return value of the function call. The only restriction is that exit codes must be integers between 0 and 255, that is, 8-bit unsigned integers. Similarly to functions in programs you write, the meaning of a specific exit code depends on the program that returns it. A rule that *all* programs follow is that an exit code of 0 indicates success and any non-zero exit code indicates a result that the caller of the program may need to be aware of; this ranges from entirely expected outcomes to serious error conditions. For normal results, different exit codes can be used in scripts to build conditional workflows that depend on the results of different steps in the script. Exit codes indicating serious errors should be handled gracefully by scripts.

The shell stores the exit code of the most recently executed program in a special variable `?`. Thus, you can display the exit code of a program `prog` after running it using the command-line

```
$ ./prog; echo $?
```

Let us try this out using the `grep` command. First use `emacs` to create a file `languages.txt`:

languages.txt

```
Rust
C
Java
C++
Python
Ruby
Haskell
Scheme
FORTRAN
```

Run the commands below to learn which exit codes `grep` uses to indicate that

- The given pattern was found, that is, there exists a line in the input that matches the pattern,
- The given pattern was not found (no line in the input matches the pattern) or
- Some serious error occurred (e.g., we tried to apply `grep` to a file that does not exist).

The first two possibilities are examples of perfectly normal outcomes; finding or not finding a match for a pattern in a file is perfectly normal, but we may want to distinguish the two outcomes programmatically, so `grep` returns different exit codes to help us with that. The third possibility is something that went wrong and `grep` returns a different exit code to distinguish such an abnormal outcome from the two expected ones.

```
$ grep C languages.txt; echo $?
```

```
$ grep LISP languages.txt; echo $?
```

```
$ grep C lenguajes.txt; echo $?
```

`search.sh`: A wrapper for `grep` with human-readable output

Now that we know `grep`'s exit codes, let us work with them in a simple shell script. Our goal is to write a simple wrapper `search.sh` around `grep` that behaves as follows:

- It does not display the lines that match the pattern if there are any; it only indicates success or failure of the search for the pattern.
- If the pattern is found, it prints `PATTERN FOUND` to `stdout`.
- If the pattern is not found, it prints `PATTERN NOT FOUND` to `stdout`.
- If an error occurrs while running `grep`, it prints `ERROR:` followed by the error message `grep` printed to `stderr`. This entire error message should be sent to `stderr`. For example with the above example of searching for the pattern `C` in the file `lenguajes.txt`, the `stderr` ouput of your script should be

  ```
  ERROR: grep: lenguajes.txt: No such file or directory
  ```

  not

  ```
  grep: lenguajes.txt: No such file or directory
  ERROR: grep: lenguajes.txt: No such file or directory
  ```

  The error message should be produced only once, prefixed with `ERROR:`.
- In all three cases above, the script should have the same exit code as the one returned by `grep`.
- If there are errors in the script itself, the script's exit code should be `3`, to distinguish this from an error in `grep`.

Step 2: Checking command-line arguments

We want `search.sh` to accept exactly two command-line arguments, the pattern to be searched for and the file to be searched. We will leave it to `grep` to deal with these arguments appropriately, so we only check that the number of arguments is correct.

Recall that `$#` is the number of command-line arguments. So let us start by writing our first version of `search.sh` that simply displays the number of given arguments. (This is useful if you don't remember whether the name of the script is counted; I never do.)

search.sh

```
#!/bin/sh
echo $#
```

Run this with a number of command-line arguments to see whether `$#` counts only the number of command-line arguments or also the command itself (it should be the former).

Now check that there are exactly two command-line arguments, print an error message if this is not the case, and, for now, simply print the command-line arguments to `stdout`. We also ensure that the wrong number of command-line arguments results in the correct exit code, 3.

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
echo $1 $2
```

Check that the exit code is 0 if you give two command-line arguments:

```
$ ./search.sh 1 2; echo $?
1 2
0
```

and that you get an error message and an exit code of 3 if you give the wrong number of command-line arguments:

```
$ ./search.sh 1 2 3; echo $?
USAGE: ./search.sh regex file
3
```

Step 3: Quoting command-line arguments

Try to run the script from Step 2 on inputs containing many spaces:

```
$ ./search.sh "   a    b   " c
a b c
```

Hmm, something is wrong; the spaces simply disappear (or rather every sequence of consecutive spaces is replaced by a single space and leading and trailing spaces are stripped).

If you think about this a little more carefully, this is not all that surprising. Command-line arguments are separated by one or more spaces. If you type

```
$ echo    a      b    c
```

you get the expected output

```
a b c
```

It should not matter how many spaces you put between command-line arguments. Now, the expression `echo $1 $2` simply means "take the variables 1 and 2 (the first and second command-line arguments) and substitute their contents in place of the expressions `$1` and `$2`". Thus, if our arguments are `"   a    b   "` and `"c"`, we simply get the code `echo   a    b   c`, which we *do* expect to print "a b c". What is even worse is that `"   a    b   "` is no longer treated as a single command-line argument. Due to the spaces it contains, it is split into two arguments, `a` and `b`; the call to `echo` has three rather than two arguments.

To check this, write a simple script

count_and_print.sh

```
#!/bin/sh
echo $#
for arg in "$@"; do
    echo "\"$arg\""
done
```

(You will learn about `$@` a little later. For now, simply take my word that this script prints its number of command-line arguments, followed by each argument in quotes, one argument per line.)

Now replace the line `echo $1 $2` in `search.sh` with `./count_and_print.sh $1 $2` and run

```
$ ./search "   a   b   " c
3
"a"
"b"
"c"
```

As expected, while `search.sh` has two arguments, they turn into three arguments "a", "b", and "c" in the argument list of `count_and_print.sh`.

What we need is a way to convince the shell to leave the command-line arguments intact. The trick is simple: use double quotes:

```
$ echo "   a    b   " c
   a    b    c
```

What is even more useful is that we can use variable references such as `$1` or `$2` inside double quotes and the shell substitutes them. Thus, if the command-line arguments are `"  a  b  "` and `"c"` as above, the expression `echo "$1" "$2"` becomes `echo "  a  b  " "c"`; echo has two arguments and produces the output

```
   a    b    c
```

To incorporate what you have learned, change your `search.sh` script to

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
./count_and_print.sh "$1" "$2"
```

and test that it behaves as expected on arguments with spaces:

```
$ ./search "a b" c
2
"a b"
"c"
```

Step 4: Single quotes vs double quotes

bash has two types of quotes with different semantics. We have seen double quotes ("...") in Step 3. Here, we will explore their difference to single quotes ('...') (not to be confused with backticks (`...`), which we used in class to capture command output). On your keyboard, single quotes are on the same key as double quotes; you get single quotes without `Shift` and double quotes with `Shift`. Backticks are on the same key as ~. Without `Shift`, you get a backtick; with shift, you get a ~.

So what is the difference? Variable substitution (and a number of other features we won't discuss here). The expression `"$var"` constructs a string whose content is the content of the variable `var`. `'$var'` produces the string `"$var"`; no replacement is performed. Let us try this out:

```
$ var='Hello, world!'
$ echo "$var"
Hello, world!
$ echo '$var'
$var
```

When do we use single quotes and when do we use double quotes? It depends on whether you want variable substitution to happen or not and whether you want special characters such as `(`, `)`, `!`, and `$` to be interpreted literally or have their special meaning in the shell.

Consider the `grep` command. Here is a use that searches for lines that contain only digits:

```
$ grep '^[0-9]*$' numbers.txt
```

Try this out after creating a file `numbers.txt` containing some lines with only numbers and some lines containing a different kind of text. It should work just fine.

Now try

```
$ grep "^[0-9]*$" numbers.txt
```

Surprisingly, this also works and running

```
$ echo "^[0-9]*$"
^[0-9]*$
```

shows that the expression does not get mangled; the `$` is not interpreted as a variable reference because it is at the end of the string. If you try the command

```
$ grep "[$9]" numbers.txt
```

to look for any lines that contain a `$` or `9`, you get

```
grep: brackets ([ ]) not balanced
```

because, with `$9` being empty, this becomes

```
$ grep "[]" numbers.txt
```

and `[]` is not a valid regular expression.

```
$ grep '[$9]' numbers.txt
```

on the other hand, works just fine. Again, you can observe what happens to your string by running

```
$ echo "[$9]"
[]
$ echo '[$9]'
[$9]
```

Step 5: The list of command-line arguments

In the case of our `search.sh` script, it seems somewhat silly to unpack the command-line arguments and pass them to `grep` individually. Once we have checked that there are exactly two command-line arguments, we just want to pass all the arguments to `grep` unchanged. This is a common enough scenario that the shell has a shorthand for it. Both `$*` and `$@` expand to the list of command-line arguments, with the same caveats as with individual arguments: If the command-line arguments are `"a b"` and `"c"`, then the line

```
$ ./count_and_print.sh $*
```

or

```
$ ./count_and_print.sh $@
```

becomes

```
$ ./count_and_print.sh a b c
```

that is, `count_and_print.sh` has three, not two, command-line arguments.

Try this out by changing the last line of your `search.sh` script first to `./count_and_print.sh $*` and then to `./count_and_print.sh $@`. In both cases, running

```
$ ./search "a b" c
```

should result in the output

```
3
"a"
"b"
"c"
```

Again, the key to keeping arguments intact is quoting, which is where `$*` and `$@` behave differently, and the behaviour of `$@` is somewhat magical.

First let us look at `$*`. Change your script so it looks like this:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
./count_and_print.sh "$*"
```

Now run your `search.sh` script using

```
$ ./search.sh "a   b" c
1
"a   b c"
```

Huh! What happens here is that the list of command-line arguments in `$*` is substituted between the quotes, separated by spaces, so we really pass a single argument `"a   b c"` to `count_and_print.sh` and the output now makes sense.

In fact, the elements of `$*` aren't always separated by spaces; that's just the default. The separator is determined by the contents of the shell variable `IFS` (input field separator). If you change `search.sh` to look like this:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
IFS=','
./count_and_print.sh "$*"
```

you should get the output

```
$ ./search.sh "a   b" c
1
"a   b,c"
```

Effectively, `"$*"` means `"$1$IFS$2$IFS..."` or, with the default value `IFS=" "`, `"$1 $2 ..."`.

What we need is a magic construct that takes the list of command-line arguments and produces a command list `"$1" "$2" ...` that quotes each argument individually. This is where `$@` comes to our help. The expression `"$@"` behaves exactly like this; it expands to `"$1" "$2" ...`.

Change `search.sh` so it looks like this:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
./count_and_print.sh "$@"
```

and test that the command-line arguments are now passed to `count_and_print.sh` unaltered:

```
$ ./search.sh "a   b" c
2
"a   b"
"c"
```

Step 6: Suppressing grep's output

Next let us run grep instead of our test script count_and_print.sh and let us for now test that grep's exit code is what we want:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
grep "$@"
echo $?
```

The problem with this script is that we still get all of grep's output, both the normal output and error messages:

```
$ ./search.sh C languages.txt
C
C++
0
$ ./search.sh LISP languages.txt
1
$ ./search.sh C lenguajes.txt
grep: lenguajes.txt: No such file or directory
2
```

We can silence grep using its -q command-line option. This does not silence error messages, exactly what we want because we want to capture them. After changing the line `grep "$@"` in your search.sh script to `grep -q "$@"`, you should get the following results:

```
$ ./search.sh C languages.txt
0
$ ./search.sh LISP languages.txt
1
$ ./search.sh C lenguajes.txt
grep: lenguajes.txt: No such file or directory
2
```

Step 7: Using exit codes

For now, let us ignore how we deal with grep's error messages correctly. Let us focus on printing whether a match was found. To do this, we need to check grep's exit code and print the correct message as a result. Let us use bash's case construct for this:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
grep -q "$@"
case $? in
    0)
        echo "PATTERN FOUND"
        ;;
    1)
        echo "PATTERN NOT FOUND"
        ;;
    *)
        echo "AN ERROR OCCURRED"
        ;;
esac
```

The * branch uses a wildcard to match any exit code not matched by the first two branches, which we know are error codes in this case.

Try it out:

```
$ ./search.sh C languages.txt
PATTERN FOUND
$ ./search.sh LISP languages.txt
PATTERN NOT FOUND
$ ./search.sh C lenguajes.txt
grep: lenguajes.txt: No such file or directory
AN ERROR OCCURRED
```

Step 8: Capturing the error messages

Now let us address how to deal with grep's error messages. We need to capture grep's output and later pass it to echo. We know a tool to do this: backticks. However, `` `grep -q "$@"` `` does not work; it captures stdout, which we already suppressed using the -q flag. What we want to capture is stderr. Using output redirection, this is easy: `` `grep -q "$@" 2>&1` ``.

So, our approach now is to capture grep's stderr using the above recipe. If there was no error, we print PATTERN (NOT) FOUND. Otherwise, we print ERROR: followed by the captured output of grep and in fact send this to stderr. This leads to the following version of search.sh:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
errors=`grep -q "$@" 2>&1`
case $? in
    0)
        echo "PATTERN FOUND"
        ;;
    1)
        echo "PATTERN NOT FOUND"
        ;;
    *)
        echo "ERROR: $errors" 1>&2
        ;;
esac
```

Try it out:

```
$ ./search.sh C languages.txt
PATTERN FOUND
$ ./search.sh LISP languages.txt
PATTERN NOT FOUND
$ ./search.sh C lenguajes.txt
ERROR: grep: lenguajes.txt: No such file or directory
```

Whoohoo!

Test that the output in the first two cases is sent to stdout and the output in the third case is sent to stderr. One way to do this is to redirect stdout and then stderr to /dev/null. This shows only the output of the channel that is *not* redirected to /dev/null.

Step 9: Returning the right exit code

Now let us address the final piece of the puzzle: returning the right exit code. Let us start by doing the natural thing: we simply return the exit code:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
errors=`grep -q "$@" 2>&1`
case $? in
    0)
        echo "PATTERN FOUND"
        ;;
    1)
        echo "PATTERN NOT FOUND"
        ;;
    *)
        echo "ERROR: $errors" 1>&2
        ;;
esac
exit $?
```

Try it out:

```
$ ./search.sh C languages.txt; echo $?
PATTERN FOUND
0
$ ./search.sh LISP languages.txt; echo $?
PATTERN NOT FOUND
0
$ ./search.sh C lenguajes.txt; echo $?
ERROR: grep: lenguajes.txt: No such file or directory
0
```

That's not what we expected. Why is the exit code always 0?

No matter the original exit code of `grep`, each branch of the `case` statement executes `echo` and `$?` becomes the exit code of this invocation of `echo`; `grep`'s exit code is lost.

The solution is simple: we need to save `grep`'s exit code before it gets overwritten by the next command:

search.sh

```
#!/bin/sh
if (( $# != 2 )); then
    echo "USAGE: $0 regex file"
    exit 3
fi
errors=`grep -q "$@" 2>&1`
result=$?
case $result in
    0)
        echo "PATTERN FOUND"
        ;;
    1)
        echo "PATTERN NOT FOUND"
        ;;
    *)
        echo "ERROR: $errors" 1>&2
        ;;
esac
exit $result
```

Finally, we get exactly the output we want:

```
$ ./search.sh C languages.txt; echo $?
PATTERN FOUND
0
$ ./search.sh LISP languages.txt; echo $?
PATTERN NOT FOUND
1
$ ./search.sh C lenguajes.txt; echo $?
ERROR: grep: lenguajes.txt: No such file or directory
2
$ ./search.sh; echo $?
USAGE: ./search.sh regex file
3
```

Step 10: Simulating grep using sed

For our next project in this lab, we need a command we did not use before: sed. Its name means "Stream EDitor"; sed treats every input file as a stream of lines and manipulates each line using a series of commands. A full exploration of sed is beyond the scope of this course (as always, you are encouraged to find more information online and play with it to learn what a powerful tool sed truly is). Here, we will explore it just enough to perform the simple transformations you will need.

Every sed command consists of two parts: a *pattern* followed by a *command*. The logic is that the command is applied to every line that matches the pattern.

The pattern is either a expression enclosed in `/.../` or it is empty, in which case the command is applied to *every* line in the input.

There are many commands, including branching instructions that allow you to write fairly complex programs in sed. Here, we consider only three such commands: `p` (print), `d` (delete), and `s` (substitute).

First, let us try to write our own grep command using sed. Here is the script you should create:

sed_grep.sh

```
#!/bin/sh
sed -e "/$1/p" $2
```

(For brevity, we do not check that there are exactly two arguments. We simply pass them to sed.)

The logic is: For every line that matches the first command-line argument interpreted as a regular expression (`/$1/`), execute the print command (`p`). Apply this to the second argument, a file.

Try this out:

```
$ ./sed_grep.sh C languages.txt
Rust
C
C
Java
C++
C++
Python
Ruby
Haskell
Scheme
FORTRAN
```

Not quite what we expected. The output contains all lines in the input and contains a duplicate of every line that matches the pattern.

The reason is that sed's default behaviour is to apply all commands whose patterns match the current line and then output the resulting line. Here, it applies the command `p` to the current line, which prints it to `stdout` but does not alter the current line. Then, after doing this, it prints the current line, which sends the line to the output a second time. If the line does not match the pattern, the `p` command is not executed but the line is still printed once because sed prints every line it processes.

We have at least two ways to ensure that *only* the lines matching the pattern are printed. The first is to disable sed's default behaviour of always printing the current line when it has been processed. sed's `-n` flag does this:

sed_grep.sh

```
#!/bin/sh
sed -n -e "/$1/p" $2
```

```
$ ./sed_grep.sh C languages.txt
C
C++
```

We can also try to use sed's delete (`d`) command to delete the lines that do not match the pattern and, as a result, print only those that do match the pattern.

To be clear, `d` deletes the current line and does not produce any output whenever the given pattern matches. Let us try this:

sed_grep.sh

```
#!/bin/sh
sed -e "/$1/d" $2
```

```
$ ./sed_grep.sh C languages.txt
Rust
Java
Python
Ruby
Haskell
Scheme
FORTRAN
```

This result should not come as a surprise. We now delete all the lines that match the search pattern and print all the remaining lines, exactly the opposite behaviour from what we want.

Can we convince sed to apply a command when the pattern *does not* match? Yes. We negate the pattern by following it with an exclamation mark:

sed_grep.sh

```
#!/bin/sh
sed -e "/$1/!d" $2
```

```
$ ./sed_grep.sh C languages.txt
C
C++
```

Step 11: Transforming lines using sed

Next, let us explore sed's substitute command (`s`). We will explore the task of the 4th question of the 1st assignment. The task was to take a CSV file like this one:

spreadsheet.csv

```
Subaru,Forester,$43000
Toyota,Corolla,$29000
Mazda,5,$38000
```

and translate it to

```
$43000,Subaru,Forester
$29000,Toyota,Corolla
$38000,Mazda,5
```

We want to apply this transformation to *every* line of the file, so there is no pattern, only the command. The `s` command is followed by an expression of the form `/pattern/replacement/`. If `s` finds a substring on the current line that matches the pattern, it greedily finds the longest such pattern and replaces it with `replacement`. `replacement` can include references to parenthesized subexpressions in `pattern`, which are replaced by the parts of the input line that match these subexpressions. The syntax is `\1`, `\2`, ... analogously to extended regular expressions.

So what do we want to do here: Any line that contains a comma should be split into the portion before the last comma and the portion after the last comma and then these two portions should be swapped. Since sed finds matches greedily, the pattern `\(.*\),\(.*\)` ensures that `\1` is the longest prefix that is succeeded by a comma and `\2` is the part that succeeds the last comma. (Yes, sadly, sed switches the semantics we are used to and requires us to escape parentheses and various special characters if we want them to have their special meaning; without escaping, parentheses and many special characters simply match themselves.) To swap these two parts, we choose our replacement to be `\2,\1`. Let us put it together:

```
$ sed -e 's/\(.*\),\(.*\)/\2,\1/' spreadsheet.csv
$43000,Subaru,Forester
$29000,Toyota,Corolla
$38000,Mazda,5
```

Step 12: Replacing all occurrences of a pattern

By default, sed's `s` command finds only the first occurrence of the pattern, greedily finds the longest match that starts at this position, and replaces it with the replacement. Let us try to figure out a way to change the field separators in `spreadsheet.csv` from commas to colons:

```
$ sed -e 's/,/:/' spreadsheet.csv
$43000:Subaru,Forester
$29000:Toyota,Corolla
$38000:Mazda,5
```

Not surprisingly, only the first comma in each line is replaced by a colon. If we want to replace *all* occurrences of the pattern, we need to provide a *modifier* to the `s` command. There are a number of such modifiers. Here, we only consider the `g` modifier, which makes `s` replace all occurrences of the pattern. The modifier succeeds the final backslash of the `s` command:

```
$ sed -e 's/,/:/g' spreadsheet.csv
$43000:Subaru:Forester
$29000:Toyota:Corolla
$38000:Mazda:5
```

Step 13: Querying a simple employee database — preparation

The final part of this lab looks at a slightly more complicated problem. This problem is complex enough that you would normally solve it using a more general-purpose programming language such as Java, Python, C or many others, not using a shell script. The point of this exercise is to demonstrate that it can be done using shell scripts and to demonstrate some more advanced uses of shell constructs.

Let us create a database of salary records for a (small) company. Each row represents one employee in the form "Last name:first name:position:department:salary":

employees.csv

```
Bossman:Big Boss:CEO:Management:$20,000,000
IRS:Fool The:Tax Accountant:Accounting:$120,000
Segundo:Pequeñito:Vice President:Management:$10,000,000
Hustler:John:Manager:Sales:$200,000
Hummer:Little:Junior Salesman:Sales:$60,000
Kaching:Dollar:Manager:Accounting:$300,000
Man:Money:Junior Analyst:Accounting:$70,000
Bee:Worker:Senior Salesman:Sales:$140,000
Man:Old:Senior Analyst:Accounting:$90,000
```

Our goal is to write a script `salaries.sh` that, by default, reports the total salary expenses we have.

```
$ ./salaries.sh employees.csv
$30,980,000
```

Using the `-d` option, it produces a list of salary commitments itemized by department:

```
$ ./salaries.sh -d employees.csv
Accounting: $580,000
Management: $30,000,000
Sales: $400,000
```

Throughout this part of the lab, we assume that every department name is a single word, as is the case for the `employees.csv` file above. Dealing with multi-word department names would add extra complications to our script. Given an input with multi-word department names, the script we develop here will most likely misbehave, possibly dramatically so.

Step 14: Checking command-line arguments

First let us check the command-line arguments. We want either a single command-line argument or two arguments, the first one being the option `-d`:

salaries.sh

```
usage() {
  echo "USAGE: $0 [-d] database"
  exit 1
}

global_sum() {
  echo "Summing all salaries"
}

detailed_sum() {
  echo "Summing by department"
}

case $# in
    1)
        global_sum "$1"
        ;;
    2)
        if [ $1 == "-d" ]; then
            detailed_sum "$2"
        else
            usage
        fi
        ;;
    *)
        usage
        ;;
esac
```

I snuck in an extra feature here, which we did not discuss before: We can define new shell commands (functions) using the `name() { ... }` syntax.

Here, we define a `usage` command, a `global_sum` command, and a `detailed_sum` command, which respectively print a usage message and exit with a non-zero exit code, implement the summing of all salaries, and implement the summing of salaries by department.

For now, the script does not do anything interesting yet:

```
$ ./salaries.sh employees.csv
Summing all salaries
$ ./salaries.sh -d employees.csv
Summing by department
$ ./salaries.sh
USAGE: ./salaries.sh [-d] database
$ ./salaries.sh -n employees.csv
USAGE: ./salaries.sh [-d] database
```

Step 15: Summing the lines of a file

First, let us focus on the simpler of the two functions, `global_sum`. Our solution consists of four parts:

1. Extract the salaries and discard all the other fields we do not need.
2. Turn the dollar amounts into plain numbers without commas.
3. Sum the resulting list of numbers.
4. Format the resulting sum so it becomes a dollar amount with commas separating groups of three digits again.

Step 1 is easy using `cut`:

```
$ cut -d: -f5 employees.csv
$20,000,000
$120,000
$10,000,000
$200,000
$60,000
$300,000
$70,000
$140,000
$90,000
```

For Step 2, we use `sed` to drop all dollar signs and commas:

```
$ cut -d: -f5 employees.csv | sed -e 's/[,$]//g'
20000000
120000
10000000
200000
60000
300000
70000
140000
90000
```

We can use a for-loop to iterate over the lines of this output and add these values to a running sum:

```
$ sum=0
$ for num in `cut -d: -f5 employees.csv | sed -e 's/[,$]//g'`; do \
  (( sum = $sum + $num )); done; echo $sum
30980000
```

Before moving on, let us implement this in our `global_sum` function:

salaries.sh
```
...
global_sum() {
    sum=0
    for num in `cut -d: -f5 $1 | sed -e 's/[,$]//g'`; do
        (( sum = $sum + $num ))
    done
    echo $sum
}
...
```

```
$ ./salaries.sh employees.csv
30980000
```

Now, to bring back the proper formatting. The logic is as follows: As long as we have a group of 4 digits at the end of the sum or succeeded by a comma, split off the last three digits using a comma. Doing this once is easy. Here we use sed's extended regular expression mode (activated using the `-E` flag), which allows us to write extended regular expressions the way we are used to:

```
$ echo 30980000 | sed -E -e 's/([0-9])([0-9]{3})($|,)/\1,\2\3/'
30980,000
```

We cannot do more than this using a single replacement. However, we can use two sed commands we have not discussed yet to repeat this replacement until we have properly split the number into groups of 3 digits: The `:` command introduces a label. `t` jumps to a given label if, since the last reading of an input line or the last execution of a `t` command, a substitution has been made. We can use this as follows: Try to apply the above replacement. If this succeeds, there is a chance that more replacements are possible. Thus, we should try to repeat the replacement by jumping back to the top of the command list (using `t`) and do this until our attempt to replace the pattern fails. If the replacement fails, then `t` does not perform the jump, so the loop ends:

```
$ echo 30980000 | sed -E -e ':loop' -e 's/([0-9])([0-9]{3})($|,)/\1,\2\3/' -e 'tloop'
30,980,000
```

With this, we can construct our final version of the `global_sum` function:

salaries.sh
```
...
global_sum() {
    sum=0
    for num in `cut -d: -f5 $1 | sed -e 's/[,$]//g'`; do
        (( sum = $sum + $num ))
    done
    sum=`echo $sum | \
         sed -E -e ':loop' -e 's/([0-9])([0-9]{3})($|,)/\1,\2\3/' -e 'tloop'`
    echo "\$$sum"
}
...
```

```
$ ./salaries.sh employees.csv
$30,980,000
```

Step 16: Summing salaries by department

To sum salaries by department, we use the following strategy. There are more efficient (and more elegant) ways to do this, but the method we use here is a reasonable solution.

1. First we extract the department and salary from each line.
2. Next we extract the list of department names.
3. For each department, we find all lines that match this department, sum the salaries in these lines using a similar strategy to the one used in `global_sum`, and print the result.

Here is Step 1:

```
$ bydepartment=`cat employees.csv | cut -d: -f4,5 | sed -e 's/[,$]//g'`
$ echo $bydepartment
Management:20000000 Accounting:120000 Management:10000000 Sales:200000 Sales:60000
Accounting:300000 Accounting:70000 Sales:140000 Accounting:90000
```

`cat employees.csv` displays the `employees.csv` file. Using `cut -d: -f4,5`, we extract only the 4th and 5th fields of each line. `sed -e 's/[,$]//g'` then discards all dollar signs and commas. By enclosing this in backticks, we capture the output, which we assign to the `bydepartment` variable. During this assignment, newlines are translated into spaces; hence the output of `echo` as one long line.

Now Step 2: We extract only the department fields from the entries in `bydepartment`, sort the resulting list, and use `sort`'s `-u` option to remove duplicates:

```
$ bydepartment=`cat employees.csv | cut -d: -f4,5 | sed -e 's/[,$]//g'`
$ departments=`for x in $bydepartment; do echo $x; done | cut -d: -f1 | sort -u`
$ echo $departments
Accounting Management Sales
```

Step 3: For a given department, say Accounting, we can get the list of salaries of employees in this department by extracting all entries in `$bydepartment` that match this department and extracting only the list of salaries from each matching line:

```
$ salaries=`for x in $bydepartment; do echo $x; done | \
            sed -E -e '/^Accounting:/!d' -e 's/.*://'`
$ echo $salaries
120000 300000 70000 90000
```

We can now sum these salaries in a manner similar to what we did in our `global_sum` function:

```
$ sum=0
$ for sal in $salaries; do (( sum = $sum + $sal )); done
$ echo $sum
580000
```

Now, all we need to do is put these different parts together in a single function:

salaries.sh

```
...
detailed_sum() {
    bydepartment=`cat employees.csv | cut -d: -f4,5 | sed -e 's/[,$]//g'`
    departments=`for x in $bydepartment; do echo $x; done | cut -d: -f1 | sort -u`
    for dep in $departments; do
        sum=0
        salaries=`for x in $bydepartment; do echo $x; done | \
                  sed -E -e "/^$dep:/!d" -e 's/.*://'`
        for sal in $salaries; do
            (( sum = $sum + $sal ))
        done
        sum=`echo $sum | \
             sed -E -e ':loop' -e 's/([0-9])([0-9]{3})($|,)/\1,\2\3/' -e 'tloop'`
        echo "$dep: \$$sum"
    done
}
...
```

Finally, note that there is some code duplication: the logic for summing the entries in a list and for formatting the sum is the same in `global_sum` and `detailed_sum`. Let us factor this into two functions addup and `format_dollars` to obtain our final script:

salaries.sh (part 1)

```
#!/bin/sh

usage() {
  echo "USAGE: $0 [-d] database"
  exit 1
}

addup() {
    sum=0
    for num in $@; do
        (( sum = $sum + $num ))
    done
    echo $sum
}

format_dollars() {
    echo $1 | sed -E -e ':loop' -e 's/([0-9])([0-9]{3})($|,)/\1,\2\3/' -e 'tloop'
}

global_sum() {
    salaries=`cut -d: -f5 $1 | sed -e 's/[,$]//g'`
    sum=`addup $salaries`
    sum=`format_dollars $sum`
    echo "\$$sum"
}

detailed_sum() {
    bydepartment=`cat employees.csv | cut -d: -f4,5 | sed -e 's/[,$]//g'`
    departments=`for x in $bydepartment; do echo $x; done | cut -d: -f1 | sort -u`
    for dep in $departments; do
        salaries=`for x in $bydepartment; do echo $x; done | \
                  sed -E -e "/^$dep:/!d" -e 's/.*://'`
        sum=`addup $salaries`
        sum=`format_dollars $sum`
        echo "$dep: \$$sum"
    done
}
```

salaries.sh (part 2)

```
case $# in
    1)
        global_sum "$1"
        ;;
    2)
        if [ $1 == "-d" ]; then
            detailed_sum "$2"
        else
            usage
        fi
        ;;
    *)
        usage
        ;;
esac
```

Step 17: Submitting your work

Copy the following files into the `csci2132/svn/CSID/lab4` directory, put them under Subversion control using `svn add`, and commit them using `svn commit`:

- `search.sh`
- `count_and_print.sh`
- `sed_grep.sh`
- `employees.csv`
- `salaries.sh`