Script arguments

Last time, we introduced the idea of shell scripts. Today, we endeavor to make shell scripts more useful.

First off, comments are useful in shell scripts. Comments are introduced by `#`, and anything after a `#` until the end of the line is ignored.

If you look back at our example last time, you notice that the shell script is not very useful. It is hard-wired to work on `file1` and `file2`. It would be much nicer to be able to specify as *arguments* to the script the two files to act on, just as you would with any other command. To deal with such arguments, when a script is executed, special shell variables are set up to contain the arguments passed to the script. These variables are referred to as `$1`, `$2`, `$3`, ..., for the first, second, third,... argument to the script. Hence, a nicer version of the script would be (in `test2.sh`):

```
#!/bin/bash

fgrep 'hi' $1 > tmp1
fgrep 'hello' $2 > tmp2
cat tmp1 tmp2 | sort
```

To invoke such a script, after making it executable,

```
test2.sh file1 file2
```

One thing that the script doesn't do is check whether it has the right number of arguments. Equivalently, it could do something different if it got no arguments, or only one arguments (for example, read one of the files from `stdin`). We will see in a little bit how to do that. For now, let's mention some other special variables set up by the shell when a script is invoked:

- `$#` holds the number of arguments supplied to the script
- `$0` holds the name under which the script was invoked (in the example above, it would hold `test2.sh`).

Exit codes

When a command exists, not only does it perform some input and output (maybe), but it also reports to the shell a status in the form of an *exit code*, a number typically between 0 and 255. An exit code of 0 means that the command succeeded, while a code different than 0 means that the command failed in some way. The man pages for the command will tell you precisely what exit codes can be returned by the command. For example, `grep` will return an exit code of 0 if it has found at least one match, it will return an exit code of 1 if it not found any matches, and it will return an exit code of 2 if there has been an error (for example, you grep on a file that does not exist).

There are many ways of combining commands, some of them using exit codes. I will use `bash` syntax for the time being, since your homework will use `bash`. Corresponding constructs exist for the C shell family.

- *command1* ; *command2* will execute first *command1*, and then execute *command2*. In a script, this is equivalent to putting the commands on two different lines.
- *command1* && *command2* will first execute *command1*, and if it succeeds, execute *command2*.
- *command1* || *command2* will first execute *command1* and if it fails, will execute *command2*

The exit code of a combined command is the exit code of the last command executed in the combination. Combinations can be nested. You can wrap any combination within *command* ; (the space after the *command* is important). The command `(`*command*`)` will execute *command* in a subshell.

A shell script can also return an exit code. By default, it is the exit code of the last command executed in the script. To force a script to terminate with a given exit code, you can use `exit` *n* where *n* is the exit code.

Note that when I talk about a command, I mean a whole pipeline of commands. The exit code of a pipeline is the exit code of the last command of a pipeline.

One place where exit codes are useful is in *conditional*. You can write the following command in a script (in fact, also on the command line, but it's rarely used): `if` *command1* ; `then` *command2* ; `fi`. The idea is to execute *command1*, and if it succeeds, to execute *command2*. Conditionals also allow the use of an else-clause, such as: `if` *command1* ; `then` *command2* ; `else` *command3* ; `fi`. Note that commands can be compound commands (perhaps wrapped within for disambiguation).

There are other control-flow commands available in the shell, such as while-loops and repeat-loops. I'll direct you to the man pages to learn more about those.

Tests

A very handy command is available to perform a variety of tests. It's main purpose is not to perform any input or output, but rather to test a condition and return an appropriate exit code. It is most useful when used in the test of a conditional. The syntax is simply `[ condition ]`.

The command returns an exit code of 0 if *condition* is satisfied, and an exit code different than 0 otherwise. Here are some useful conditions. Other conditions can be found on the `bash` man page.

- `-f` *file* tests if *file* exists and is regular (i.e. not a directory or a special file such as a link or a driver)
- `-r` *file* tests if *file* exists and is readable
- `-d` *file* tests if *file* exists and is a directory
- `-n` *string* tests if *string* has non-zero length
- `-z` *string* tests if *string* has zero length
- *s1* `=` *s2* tests if strings *s1* and *s2* are equal
- *n1* `-gt` *n2* tests if $n1 > n2$
- *n1* `-lt` *n2* tests is $n1 < n2$

You may want to put strings in double-quotes, especially if they arise out of substitution of shell variables, to avoid problems if the string turns out to be multiple words (or none).

For example, to check that our script above is supplied with at least two arguments, you can use:

```
if [ "$#" -lt 2 ]
then
    echo "Not enough arguments! Aborting"
    exit 1
fi
```