Conditionals

Conditional statement allow you to execute commands based on the exit code (success or failure) of other commands. The basic form of the conditional is:

```bash
if cmd1
  then
    cmd
    cmd
fi
```

The interpretation is simple: `cmd1` is first executed, and if it succeeds (if its exit code is 0), the commands between `then` and `fi` are executed. Otherwise, execution continues after `fi`. A form of the conditional with an `else` clause is available:

```bash
if cmd1
  then
    cmd
    cmd
  else
    cmd
    cmd
fi
```

as well as a form with multiple sequential tests:

```bash
if cmd1
  then
    cmd
    cmd
  elif cmd2
    then
      cmd
      cmd
```
elif cmd3
then
...
else
  cmd
cmd
fi

The interpretation of those forms should be straightforward.

It is sometimes useful to have an empty branch in an if statement. The command : is a no-op command. Hence,

if cmd1
then
  :
else
  cmd
cmd
fi

does nothing if cmd1 succeeds, and executes the commands in the else branch otherwise.

Sometimes you will want to have a look at an exit code, for example, when your if does not behave the way you want. The variable $? holds the exit code of the last command executed, in human-readable form.

**While loops**

While loops allow you to execute commands until essentially a given condition is met. The basic form of the statement is:

while cmd1
do
  cmd
cmd
done

The interpretation is as follows. First, cmd1 is executed. If it succeeds, the commands in the body of the loop (between do and done) are executed. Then, cmd1 is executed again, and if it succeeds, the body of the loop is executed again, and so on until cmd1 fails.
An alternate form allows you to loop until a given condition is met:

```
until cmd1
do
cmd
cmd
done
```

The interpretation is as in the `while` case, except that the body is executed as long as `cmd1` fails.

**Tests**

Although any command can be used to branch in conditional statements, or to control the looping in `while` loops, a special command is very useful to perform certain tests.

The command `[testexpr]` is a built-in command that performs a test specified by `testexpr`. If the test is true, then `[testexpr]` returns an exit code of 0 (i.e., it succeeds), otherwise it fails. This command is therefore useful to control conditionals and loops. Note that variable substitution and word splitting are performed on `testexpr`, but matching is not performed.

There are many possibilities for `testexpr`, and we will not describe them all. Refer to the `bash` man pages for a full description. We will describe the most commonly used here.

Testing expressions for strings include the following:

- `str1 = str2` tests if `str1` and `str2` are equal
- `str1 != str2` tests if `str1` and `str2` are not equal
- `str` tests if `str` is non-null

Typically, these tests are used in conjunction with variables, for example, to test if a given variable has a given value. Here, one is often bitten by word splitting. Consider what happens if you attempt to test whether variable `foo` has value `John`. If you try `[ $foo = John ]`, you will get a problem if `foo` is either undefined or has a null value. Recall that the shell expands the command line, performing substitutions and such. If `foo` has a null value, then after substitution, the shell will attempt to evaluate `[ = John ]`, which will give you a syntax error. What you want is the shell to still consider for something to be there even if the variable was null-valued. It turns out you can use the form `""` to represent an explicit null string (i.e. it’s a null string, but the shell sees it as such). Since the shell will perform variable substitution under double-quotes, you can therefore write `[ "$foo" = John ]` to test the variable `foo`. If `foo` is null, then this will expand to `[ "" = John ]`, which is false. In general, it is good policy to put variables under double-quotes in tests.

A fair number of testing expressions exist for testing strings. Representatives include:
-e path  tests if path exists
-d path  tests that path exists and is in fact a directory
-f path  tests that path exists and is not a directory
-r path  tests whether you have read permissions to path
-w path  tests whether you have write permissions to path
-x path  tests whether you have execute permissions to path

Boolean combinations of test expressions are allowed:

( testexpr ) tests testexpr (useful to group conditions)
testexpr1 -a testexpr2 true iff both testexpr1 and testexpr2 are true
testexpr1 -o testexpr2 true iff either testexpr1 or testexpr2 is true
! testexpr  true iff testexpr is false

Here’s an example of testing in loops. It also serves to introduce an interesting builtin bash command. The command read is used to read input from the user, something that may be useful in a shell script. The command read var will read a line of input from the user, and put the line in variable var. If more than one variable is supplied, i.e., read var1 var2 var3, then the first word of the line input from the user is put in var1, the second in var2, and the rest of the line is put in var3. A prompt may be supplied by using a -p option, as in read -p 'some test' var (notice the quotes to give a prompt which may contain spaces...) The following lines will repeatedly query the user for a yes or no until he gets it right:

read -p "yes or no? " answer
while [ "$answer" != "yes" -a "$answer" != "no" ]
do
  echo "Please enter yes or no"
  read -p "yes or no? " answer
done
echo "the answer was $answer"

I should note that bash has an alternate more modern form of testing, written [[ testexpr ]], that does not perform word splitting (and hence solving the messy null-variable problem), and allowing pattern matching. We will not cover this form here, but I will refer you to the man pages.

**For loops**

A final form of looping is available, that does not rely on exit codes. The for statement has the form:

for var in word1 word2 ... wordn
do
    cmd
    cmd
done

The interpretation is as follows. The variable *var* is assigned the first word *word1* in the list, and the commands between *do* and *done* are executed. Then, the variable *var* gets the second word in the list, and the commands are executed. So on until all the words in the list have been processed.

Consider a simple example, to mail a file *letter* to a list of people:

```bash
for name in alice bob trudy
done
```

Of course, the list of words can be generated by substitution, as in:

```bash
names="alice bob"
junk="abc"
morenames="oscar trudy"
for name in $names $junk $morenames
done
```

Some things to notice in the above example. First, after substitution in the *for* loop, the list of words contains five words. Second, the option -n to *echo* suppresses the newline that gets added at the end of what’s printed. Third, recall that *grep* returns an exit code of 0 if a match is found, and an error code of 1 if no match is found (and no error occurred). Finally, the redirection to /dev/null is used to suppress output by *grep*, which by defaults sends the matching lines to *stdout*. (/dev/null is the so-called Unix bit-bucket; it is a black holes that just swallows input.)