Recall that whenever a script is executed, the arguments passed to the script are available as the variables `$1`,`$2`,...,`$9`. The complete list of arguments (as a single token) is available as `$*`. The number of arguments is available as `$#`. The command used to invoke the script is available as `$0`.

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:

```
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:

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

as well as a form with multiple sequential tests:

```
it 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.

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

It is important to remember that in the above tests, the expressions tested are string There are a number of tests that involve arithmetical comparisons.

num1 `-eq` *num2*	tests if *num1* $=$ *num2*
num1 `-ne` *num2*	tests if *num1* $\neq$ *num2*
num1 `-lt` *num2*	tests if *num1* $<$ *num2*
num1 `-le` *num2*	tests if *num1* $\leq$ *num2*
num1 `-gt` *num2*	tests if *num1* $>$ *num2*
num1 `-ge` *num2*	tests if *num1* $\geq$ *num2*

If the expression *num1* or *num2* are not representations of integers, an error is reported. (Earlier versions of `bash` would simply treat the expression as the integer 0.)

For example, here is how to test that a script has been called with at least two arguments:

```
if [ "$#" -lt 2 ]     # this is the test
then
  echo "Not enough arguments"
  exit 1              # exit the script (with exit code 1)
fi
```

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 patttern matching. We will not cover this form here, but I will refer you to the man pages.