

Commands and exit codes

Recall that every command¹ returns an exit code to its parent process (the process that invoked it in the first place, typically the shell), a number between 0 and 255. A value of 0 indicates success (however success is defined for the command at hand), while a value greater than 0 represents failure. Every command may define its own meaning for the failure code (i.e., what does it mean when it returns 1, or 2, or 3). Some error codes are reserved for *bash*. For example, if a command returns an exit code of 127, it means that in fact *bash* was not able to execute the command because it couldn't find it, maybe because of an error in syntax. Similarly, exit codes greater than 127 indicate that the command was interrupted due to the arrival of a signal. (We will not cover signals in this course.)

Commands can be composed in different ways on the command line. The form *cmd1 ; cmd2* executes *cmd1* and then *cmd2*. The exit code returned is the exit code of *cmd2*. The form { *cmd1 ; cmd2 ;* } (the trailing *;* is required) is similar to the previous case, except that this form allows redirection to apply to all commands in the list at once. The form (*cmd1 ; cmd2*) is similar to the previous case, except that the commands are executed in a subshell. Finally, conditional execution is expressed in two different ways. The form *cmd1 && cmd2* executes *cmd1*, and if it succeeds (i.e., it returns exit code 0), then executes *cmd2*. Alternatively, *cmd1 || cmd2* executes *cmd1*, and if it fails (i.e. returns exit code > 0), then executes *cmd2*.

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
```

¹I will use the term "command" to stand for either commands builtin to the shell, programs executed from the filesystem, aliases, etc.

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:

```
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:

<code>str1 = str2</code>	tests if <i>str1</i> and <i>str2</i> are equal
<code>str1 != str2</code>	tests if <i>str1</i> and <i>str2</i> are not equal
<code>str</code>	tests if <i>str</i> 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:

<code>-e path</code>	tests if <i>path</i> exists
<code>-d path</code>	tests that <i>path</i> exists and is in fact a directory
<code>-f path</code>	tests that <i>path</i> exists and is not a directory
<code>-r path</code>	tests whether you have read permissions to <i>path</i>
<code>-w path</code>	tests whether you have write permissions to <i>path</i>
<code>-x path</code>	tests whether you have execute permissions to <i>path</i>

Boolean combinations of test expressions are allowed:

<code>(testexpr)</code>	tests <i>testexpr</i> (useful to group conditions)
<code>testexpr1 -a testexpr2</code>	true iff both <i>testexpr1</i> and <i>testexpr2</i> are true
<code>testexpr1 -o testexpr2</code>	true iff either <i>testexpr1</i> or <i>testexpr2</i> is true
<code>! testexpr</code>	true iff <i>testexpr</i> 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:

```
for name in alice bob trudy
do
    mail $name < letter
    echo "letter mailed to $name"
done
```

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

```
names="alice bob"
junk="abc"
morenames="oscar trudy"
for name in $names $junk $morenames
do
    echo -n "Processing $name: "
    if finger $name@cornell.edu | grep 'no matches' > /dev/null
    then
        echo "match not found"
    else
        echo "match found"
    fi
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 default sends the matching lines to *stdout*. (*/dev/null* is the so-called Unix bit-bucket; it is a black hole that just swallows input.)