Bourne Shell Scripting/Print Version

From Wikibooks, the open-content textbooks collection

Hi there! Welcome to this Wikibook on the wonderful world of the Bourne Shell!

This book will cover the practical aspects of using and interacting with the Bourne Shell, the root of all shells in use in the Unix world. That includes interacting with the shell on a day-to-day basis for the purposes of operating the computer in normal tasks, as well as grouping together commands in files (scripts) which can be run over and over again. Since it's not practical to talk about the Bourne Shell in complete isolation, this will also mean some short jaunts into the wondrous world of Unix; not far, just enough to understand what is going on and be able to make full use of the shell's very wide capabilities.

There are also some things this book **won't** do for you. This book is not an in-depth tutorial on any kind of programming theory -- you won't learn the finer points of program construction and derivation or the mathematical backings of program development here. This book also won't teach you about Linux or any other type of Unix or Unix itself or any other operating system any more than is necessary to teach you how to use the shell. Nothing to be found here about Apache, joe, vi, or any other specific program. Nor will we cover firewalls and networking.

We *will* cover the Bourne Shell, beginning with the basic functionality and capabilities as they existed in the initial release, through to the added functionality specified by the international POSIX standard POSIX 1003.1 for this shell. We will have to give you *some* programming knowledge, but we hope that everyone will readily understand the few simple concepts we explain.

Having said that, the authors hope you will find this book a valuable resource for learning to use the shell and for using the shell on a regular basis. And that you might even have some fun along the way.

Authors

1. BenTels, started the book
2. Kernigh, added Substitution and Loops chapters
3. Quick reference was originally by Greg Goebel and was from http://www.vectorsite.net/tsshell.html (was public domain licensed) and was partly wikified by unforgettableid.
4. Other and anonymous contributors

External References

- IEEE Std 1003.1, 2004 Edition (http://www.unix.org/version3/online.html) - The 2004 Edition of IEEE/POSIX standard 1003.1 (one-time rgistration required).
- An Introduction to the Unix Shell (http://steve-parker.org/sh/bourne.shtml) - HTML format republication of Steve Bourne's original tutorial for the Bourne Shell.
- UNIX Shell Script Tutorials & Reference (http://www.injunea.demon.co.uk/pages/page201.htm)
- **Beginner**
  - BASH Programming - Introduction HOWTO (http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html)
  - Linux Shell Scripting Tutorial - A Beginners handbook (http://bash.cyberciti.biz/guide/Main_Page)
- **Advanced scripting guide**
  - Advanced Bash-Scripting Guide (http://tldp.org/LDP/abs/html/index.html)
- **Print**
  - UNIX IN A NUTSHELL: A Desktop Quick Reference for System V & Solaris 2.0 (2nd edition)
    Daniel Gilly et al.
    August 1994
    ISBN 1-56592-001-5

Comparing Shells

Almost all books like this one have a section on (or very similar to) "why you should use the shell/program flavor/language/etc. discussed in this book and not any of the others that perform the same tasks in a slightly different way". It seems to be pretty well mandatory.

However, this book will not do that. We'll talk a bit about "why Bourne Shell" of course. But you'll soon see that doesn't preclude other shells at all. And there's no good reason not to use another shell either, as we will explain a little further down.

Bourne shell and other Unix command shells

There are many Unix command shells available today. Bourne Shell is just one drop in a very large ocean. How do all these shells relate? Do they do the same things? Is one better than the other? Let's take a look at what makes a shell and what that means for all the different shells out there.

How it all got started...

The Unix operating system has had a unique outlook on the world and the correct way of doing things ever since it was created back in the 1970s. One of the things that sets Unix apart from most other operating systems is that it has always been an OS whose primary focus was on what we call power users nowadays: people who want to squeeze every drop out of their system and know how. People who don't sit around and dream of what could have been, but crank up the compiler and start coding. Unix was meant to be programmed on. Unix is nothing if not a platform that you have to program on to use it. Unix doesn't have a user's interface -- Unix has a stable OS kernel and a C library. If you're not trying to do actual hard-core programming but rather are trying to do day-to-day tasks (or even just want to put a little program together quickly), pure Unix is a tremendous pain in the backside.

In other words, it was clear from the start that a tool would be needed that would allow a user to make use of the functions offered him by the coding library and kernel without actually doing serious programming work. A tool in other words that could pass small commands on to the lower-level system quickly and easily. Without the need for a compiler or other fancy editor, but with the ability to tap into the enormous power of the underlying system. Stephen Bourne set himself to the task and came up with what he called a *shell*: a small, on-the-fly compiler that could take one command at a time, translate it into the sequence of bits understood by the machine and have that command carried out. We tend to call such a program an interpreter nowadays, but back then shell seemed a better word (since it was a shell over the underlying system for the user). Stephen's shell was small, slim, fast, a bit unwieldy at times and it oozed raw power that is still the envy of many an operating system's command-line interface. Since it was designed by Stephen Bourne, this shell is called the Bourne Shell. Its executable was called simply *sh* and use of this shell in scripting is still so ubiquitous that there isn't a Unix-based system on this earth that doesn't offer a shell whose executable can be reached under the name sh.

...And how it ended up

Of course, everyone's a critic. The Bourne Shell saw tremendous use (indeed, it still does) and as a result, it became the de facto standard among Unix shells. But all sorts of people almost immediately (as well as with use) wanted new features in the shell, or a more familiar way of expressing commands, or something else. Many people built new shells that they felt continued where Bourne Shell ended. Some were completely compatible with Bourne Shell, others were less so. Some became famous, others flopped. But pretty much all of them look fondly upon Bourne Shell, the shell they call "Dad..."

A number of these shells can be run in sh-like mode, to more closely emulate that very first sh, though most people tend just to run their shells in the default mode, which provides more power than the minimum sh.

It's Bourne Shell, but not as we know it....

So there are a lot of shells around but you can find Bourne Shell everywhere, right? Good old *sh*, just sitting there faithfully until the end of time....

Well, no, not really. Most of the sh exectuables out there nowadays aren't really the Bourne Shell anymore. Through a bit of Unix magic called a link (which allows one file to masquerade as another) the sh executable you find on any Unix system is likely actually to be one of the shells that is based on the Bourne shell. One of the most frequently used shells nowadays (with the ascent of free and open-source operating systems like Linux and

FreeBSD) is a heavily extended form of the Bourne Shell produced by the Free Software Foundation, called Bash. Bash hasn't forgotten its roots, though: it stands for the **B**ourne **A**gain **SH**ell.

Another example of a descendant shell standing in for its ancestor is the Korn Shell (ksh). Also an extension shell, it is completely compatible with sh -- it simply adds some features. Much the same is true for zsh.

Finally, a slightly different category is formed by the C Shell (csh) and its descendant tcsh. These shells do break compatibility to some extent, using different syntax for many commands. Systems that use these shells as standard shells often provide a real Bourne Shell executable to run generic Bourne Shell scripts.

Having read the above, you will understand why this book doesn't have to convince you to use Bourne Shell instead of any other shell: in most cases, there's no noticeable difference. Bourne Shell and its legacy have become so ingrained in the heart and soul of the Unix environment that you are using Bourne Shell when you are using practically any shell available to you.

Why Bourne Shell

So only one real question remains: now that you find yourself on your own, cozy slice of a Unix system, with your own shell and all its capabilities, is there any real reason to use Bourne Shell rather than using the whole range of your shells capabilities?

Well, it depends. Probably, there isn't. For the most part of course, you *are* using Bourne Shell by using the whole potential of your shell -- your shell is probably *that* similar to the Bourne Shell. But there is one thing you might want to keep in mind: someday, you might want to write a script that you might want to pass around to other people. Of course you can write your script using the full range of options that your shell offers you; but then it might not work on another machine with another shell. This is where the role of Bourne Shell as the Latin of Unix command shells comes in -- and also where it is useful to know how to write scripts targeted specifically at the Bourne Shell. If you write your scripts for the Bourne Shell and nothing but the Bourne Shell, chances are far better than equal that your script will run straight out of the mail attachment (don't tell me you're still using boxes to ship things -- come on, get with the program) on any command shell out there.

Running Commands

Before we can start any kind of examination of the abilities of the Bourne Shell and how you can tap into its power, we have to cover some basic ground first: we have to discuss how to enter commands into the shell for execution by that shell.

The easy way: the interactive session

Taking another look at what you've probably already seen

If you have access to a Unix-based machine (or an emulator on another operating system), you've probably been using the Bourne Shell -- or one of its descendants -- already, possibly without realising. Surprise: you've been doing shell scripting for a while already!

In your Unix environment, go to a terminal; either a textual logon terminal, or a terminal-in-a-window if you're using the X Window System (look for something called *xterm* or *rxvt* or just *terminal*, if you have actually not ever done this yet). You'll probably end up looking at a screen looking something like this:

```
Ben_Tels:Local_Machine:~>_
```

or

```
The admin says: everybody, STOP TRYING TO CRASH THE SYSTEM
Have a lot of fun!
bzt:Another_Machine:~>_
```

or even something as simple as

```
$_
```

That's it. That's your shell: your direct access to everything the system has to offer.

Using the shell in interactive mode

Specifically, the program you accessed a moment ago is your shell, running in *interactive mode*: the shell is running in such a way that it displays a prompt and a cursor (the little, blinking line) and is waiting for you to enter a command for it to execute. You execute commands in interactive mode by typing them in, followed by a press of the **Enter** key. The shell then translates your command to something the operating system understands and passes off control to the operating system so that it can actually carry out the task you have sent it. You'll notice that your cursor will disappear momentarily while the command is being carried out, and you cannot type anymore (at this point, the Bourne Shell program is no longer in control of your terminal -- the other program that you started by executing your command is). At some point the operating system will be finished working on your command and the shell will bring up a new prompt and the cursor as well and will then start waiting again for you to enter another command. Give it a try: type the command

```
ls enter
```

After a short time, you'll see a list of files in the working directory (the directory that your shell considers the "current" directory), a new prompt and the cursor.

This is the simplest way of executing shell commands: typing them in one at a time and waiting for each to complete in order. The shell is used in this way very often, both to execute commands that belong to the Bourne Shell programming language and simply to start running other programs (like the ls program from the example above).

A useful tidbit

Before we move on, we'll mention two useful key combinations when using the shell: the command to interrupt running programs and shell commands and the command to quit the shell (although, why you would ever want to *stop* using the shell is beyond me....).

To interrupt a running program or shell command, hit the Control and C keys at the same time. We'll get back to what this does exactly in a later chapter, but for now just remember this is the way to interrupt things.

To quit the shell session, hit Control+d. This key combination produces the Unix end-of-file character -- we'll talk more later about why this also terminates your shell session. Some modern shells have disabled the use of Control+d in favor of the "exit" command (shame on them). If you're using such a shell, just type the word "exit" (like with any other command) and press Enter (from here on in, I'll leave the "Enter" out of examples).

The only slightly less easy way: the script

As we saw in the last section, you can very easily execute shell commands for all purposes by starting an interactive shell session and typing your commands in at the prompt. However, sometimes you have a set of commands that you have to repeat regularly, even at different times and in different shell sessions. Of course, in the programming-centric environment of a Unix system, you can write a program to get the same result (in the C language for instance). But wouldn't it be a lot easier to have the convenience of the shell for this same task? Wouldn't it be more convenient to have a way to replay a set of commands? And to be able to compose that set as easily as you can write the single commands that you type into the shell's interactive sessions?

The shell script

Fortunately, there *is* such a way: the Bourne Shell's *non-interactive* mode. In this mode, the shell doesn't have a prompt or wait for your commands. Instead, the shell reads commands from a text file (which tells the shell what to do, kind of like an actor gets commands from a script -- hence, shell script). This file contains a sequence of commands, just as you would enter them into the interactive session at the prompt. The file is read by the shell from top to bottom and commands are executed in that order.

A shell script is very easy to write; you can use any text-editor you like (or even any wordprocessor or other editor, as long as you remember to save your script in plain text format). You write commands just as you would in the interactive shell. And you can run your script the moment you have saved it; no need to compile it or anything.

Running a shell script

To run a shell script (to have the shell read it and execute all the commands in the script), you enter a command at an interactive shell prompt as you would when doing anything else (if you're using a graphical user interface, you can probably also execute your scripts with a click of the mouse). In this case, the program you want to start is the shell program itself. For instance, to run a script called *MyScript*, you'd enter this command in the interactive shell (assuming the script is in your working directory):

Running a script

```
sh MyScript
```

Starting the shell program from inside the shell program may sound weird at first, but it makes perfect sense if you think about it. After all, you're typing commands in an *interactive mode* shell session. To run a script, you want to start a shell in *non-interactive mode*. That's what's happening in the above command. You'll note that the Bourne Shell executable takes a single parameter in the example above: the name of the script to execute.

If you happen to be using a POSIX 1003.1-compliant shell, you can also execute a single command in this new, non-interactive session. You have to use the -c command-line switch to tell the shell you're passing in a command instead of the name of a script:

Running a command in a new shell

```
sh -c ls
```

We'll get to why you would want to do this (rather than simply enter your command directly into the interactive shell) a little further down.

There is also another way to run a script from the interactive shell: you type the execute command (a single period) followed by the name of the script:

Sourcing a script

```
. MyScript
```

The difference between that and using the *sh* command is that the *sh* command starts a new process and the execute command does not. We'll look into this (and its importance) in the next section. By the way, this notation with the period is commonly referred to as *sourcing* a script.

Running a shell script the other way

There is also another way to execute a shell script, by making more direct use of a feature of the Unix operating system: the executable mode.

In Unix, each and every file has three different permissions (read, write and execute) that can be set for three different entities: the user who owns the file, the group that the file belongs to and "the world" (everybody else). Give the command

```
ls -l
```

in the interactive shell to see the permissions for all files in the working directory (the column with up to nine letters, r, w and x for read write and execute, the first three for the user, the middle ones for the group, the right ones for the world). Whenever one of those entities has the "execute" permission, that entity can simply run the file as a program. To make your scripts executable by everybody, use the command

```
chmod +x scriptname
```

as in

Making MyScript executable

```
chmod +x MyScript
```

You can then execute the script with a simple command like so (assuming it is in a directory that is in your PATH, the directories that the shell looks in for programs when you don't tell it exactly where to find the program):

Running a command in a new shell

```
MyScript
```

If this fails then the current directory is probably not in your PATH. You can force the execution of the script using

Making the shell look for your script in the current directory

```
./MyScript
```

At this command, the operating system examines the file, places it in memory and allows it to run like any other program. Of course, not every file makes *sense* as a program; a binary file is not necessarily a set of commands that the computer will recognize and a text file cannot be read by a computer at all. So to make our scripts run like this, we have to do something extra.

As we mentioned before, the Unix operating system starts by examining the program. If the program is a text file rather than a binary one (and cannot simply be executed), the operating system expects the first line of the file to name the interpreter that the operating system should start to interpret the rest of the file. The line the Unix operating system expects to find looks like this:

```
#!full path and name of interpreter
```

In our case, the following line should work pretty much everywhere:

```
#!/bin/sh
```

The Bourne Shell executable, to be found in the bin directory, which is right under the top of the filesystem tree. For example:

Bourne shell script with an explicit interpreter

Code:

```
#!/bin/sh
echo Hello World!
```

Output:

```
Hello World!
```

Executing shell scripts like this has several advantages. First it's less cumbersome than the other notations (it requires less typing). Second, it's an extra safety if you're going to pass your scripts around to others. Instead of relying on them to have the right shell, you can simply specify which shell they should use. If Bourne Shell is enough, that's what you ask for. If you asolutely need *ksh* or *bash*, you specify that instead (mind you, it's not foolproof — other people can ignore your interpreter specification by running your script with one of the other commands that we discussed above, even if the script probably won't work if they do that).

Just as a sidenote, Unix doesn't limit this trick to shell scripts. Any script interpreter that expects its scripts to be plain-text can be specified in this way. You can use this same trick to make directly executable Perl scripts or Python, Ruby, etc. scripts as well as Bourne Shell scripts.

A little bit about Unix and multiprocessing

Why you want to know about multiprocessing

While this is not directly a book about Unix, there are some aspects of the Unix operating system that we must cover to fully understand why the Bourne Shell works the way it does from time to time.

One of the most important aspects of the Unix operating system – in fact, the main aspect that sets it apart from all other main-stream operating systems – is that the Unix Operating System is and always has been a multi-user, multi-processing operating system (this in contrast with other operating systems like MacOS and Microsoft's DOS/Windows operating systems). The Unix OS was always meant to run machines that would be used simultaneously by several users, who would all want to run at least one but possibly several programs at the same time. The ability of an operating system to divide the time of a machine's processor among several programs so that it seems to the user that they are all running at the same time is called *multiprocessing*. The Unix Operating System was designed from the core up with this possibility in mind and it has an effect on the way your shell sessions behave.

Whenever you start a new process (by running a program, for instance) on your Unix machine, the operating system provides that process with its very own operating environment. That environment includes some memory for the process to play in and it can also include certain predefined settings for all processes. Whenever you run the shell program, it is running in its own environment.

Whenever you start a new process from another process (for instance by issuing a command to your shell program in interactive mode), the new process becomes what is called a *child process* of the first process (the ls program runs as a child process of your shell, for instance). This is where it becomes important to know about multiprocessing and process interaction: a child process always starts with a *copy* of the environment of the parent process. This means two things:

1. a child process can *never* make changes to the operating environment of its parent -- it only has access to a copy of that environment
2. if you actually do *want* to make changes in the environment of your shell (or specifically want to avoid it), you have to know when a command runs as a child process and when it runs within your current shell; you might otherwise pick a variant that has the opposite effect of that which you want

What does what

We have seen several ways of running a shell command or script. With respect to multiprocessing, they run in the following way:

Way of running	Runs as
Interactive mode command	<ul><li>current environment for a shell command [1])</li><li>child process for a new program</li></ul>
Shell non-interactive mode	child process
Dot-notation run command (*. MyScript*)	current environment
Through Unix executable permission with interpreter selection	child process

A useful thing to know: background processes

With the above, it may seem like multiprocessing is just a pain when doing shell scripting. But if that were so, we wouldn't *have* multiprocessing -- Unix doesn't tend to keep things that aren't useful. Multiprocessing is a valuable tool in interacting with the rest of the system and one that you can use to work more efficiently. There are many books available on the benefits of multiprocessing in program development, but from the point of view of the Bourne Shell user and scripter the main one is the ability to hand off control of a process to the operating system *and still keep on working while that child process is running*. The way to do this is to run your process as a *background process*.

Running a process as a background process means telling the operating system that you want to start a process, but that it should not attach itself to any of the interactive devices (keyboard, screen, etc.) that its parent process is using. And more than that, it also tells the operating system that the request to start this child process should return immediately and that the parent process should then be allowed to continue working without having to wait for its child process to end.

This sounds complicated, but you have to keep in mind that this ability is completely ingrained in the Unix operating system and that the Bourne Shell was intended as an easy interface to the power of Unix. In other words: the Bourne Shell includes the ability to start a child process as a simple command of its own. Let's demonstrate how to do this and how useful the ability is at the same time, with an example. Give the following (rather pointless but still time consuming) command at the prompt:

```
N=0 && while [ $N -lt 10000 ]; do date >> scriptout; N=`expr $N + 1`; done
```

We'll get into what this says in later chapters; for now, it's enough to know that this command asks the system for the date and time and writes the result to a file named "scriptout". Since it then repeats this process 10000 times, it may take a little time to complete.

Now give the following command:

```
    N=0 && while [ $N -lt 10000 ]; do date >> scriptout; N=`expr $N + 1`; done&
```

You'll notice that you can immediately resume using the shell (if you don't see this happening, hit Control+C and check that you have the extra ampersand at the end). After a while the background process will be finished and the scriptout file will contain another 10000 time reads.

The way to start a background process in Bourne Shell is to append an ampersand (&) to your command.

Remarks

^ Actually, you can force a child process here as well -- we'll see how when we talk about command grouping

Environment

No program is an island unto itself. Not even the Bourne Shell. Each program executes within an *environment*, a system of resources that controls how the program executes and what external connections the program has and can make. And in which the program can itself make changes.

In this module we discuss the environment, the habitat in which each program and command lives and executes. We look at what the environment consists of, where it comes from and where it's going... And we discuss the most important mechanism that the shell has for passing data around: the *environment variable*.

The Environments

When discussing a Unix shell, you often come across the term "environment". This term is used to describe the context in which a program executes and is usually meant to mean a set of "environment variables" (we'll get to those shortly). But in fact there are two different terms that are somehow a program's environment and which often get mixed up together in "environment". The simpler one of these really is the collection of environment variables and actually is called the "environment". The second is a much wider collection of resources that influence the execution of a program and is called the *command execution environment*.

The command execution environment

Each running program, either started directly by the user from the shell or indirectly by another process, operates within a collection of global resources called its **command execution environment** (CEE).

A program's CEE contains important information such as the source and destination of data upon which the program can operate (also known as the standard input, standard output and standard error handles). In addition, variables are defined that list the identity and home directory of the user or process that started the program, the hostname of the machine and the kind of terminal used to start the program. There are other variables too, but that's just some of the main ones. The environment also provides working space for the program, as well as a simple way of communicating with other, future programs, that will be run in the same environment.

The complete list of resources included in the shell's CEE is:

- Open files held in the parent process that started the shell. These files are inherited. This list of files includes the files accessed through redirection (such as standard input, output and error files).
- The current working directory: the "current" directory of the shell.
- The file creation mode:The default set of file permissions set when a new file is created.
- The active traps.
- Shell parameters and variables set during the call to the shell or inherited from the parent process.
- Shell functions inherited from the parent process.
- Shell options set by *set* or *shopts*, or as command line options to the shell executable.
- Shell aliases (if available in your shell).
- The process id of the shell and of some processes started by the parent process.

Whenever the shell executes a command that starts a child process, that command is executed it its own CEE. This CEE inherits a copy of part of the CEE of its parent, but not the entire parent CEE. The inherited copy includes:

- Open files.
- The working directory.
- The file creation mode mask.
- Any shell variables and functions that are marked to be exported to child processes.
- Traps set by the shell.

The 'set' command

The 'set' command allows you to set or disable a number of options that are part of the CEE and influence the behavior of the shell. To set an option, set is called with a command line argument of '-' followed by one or more flags. To disable the option, set is called with '+' and then the same flag. You probably won't use these options very often; the most common use of 'set' is the call without any arguments, which produces a list of all defines names in the environment (variables and functions). Here are some of the options you might get some use out of:

+/-a
: When set, automatically mark all newly created or redefined variables for export.

+/-f
: When set, ignore filename metacharacters.

+/-n
: When set, only read commands but do not execute them.

+/-v
: When set, causes the shell to print commands as they are read from input (verbose debugging flag).

+/-x
: When set, causes the shell to print commands as they will be executed (debugging flag).

Again, you'll probably mostly use set without arguments, to inspect the list of defined variables.

The environment and environment variables

Part of the CEE is something that is simply called the *environment*. The environment is a collection of name/value pairs called *environment variables*. These variables technically also contain the shell functions, but we'll discuss those in a separate module.

An environment variable is a piece of labelled storage in the environment, where you can store anything you like as long as it fits. These spaces are called variables because you can vary what you put in them. All you need to know is the name (the label) that you used for storing the content. The Bourne shell also makes use of these "environment variables". You can make scripts that examine these variables, and those scripts can make decisions based on the values stored in the variables.

An environment variable is a name/value pair of the form

```
name=variable
```

which is also the way of creating a variable. There are several ways of using a variable which we will discuss in the module on substitution, but for now we will limit ourselves to the simple way: if you prepend a variable name with a $-character, the shell will substitute the value for the variable. So, for example:

Simple use of a variable

Code:

```
$ VAR=Hello
$ echo $VAR
```

Output:

```
Hello
```

As you can see from the example above, an environment variable is sort of like a bulletin board: anybody can post any kind of value there for everybody to read (as long as they have access to the board). And whatever is posted there can be interpreted by any reader in whatever way they like. This makes the environment variable a very general mechanism for passing data along from one place to another. And as a result environment variables are used for all sorts of things. For instance, for setting global parameters that a program can use in its execution. Or for setting a value from one shell script to be picked up by another. There are even a number of environment variables that the shell itself uses in its configuration. Some typical examples:

IFS
: This variable lists the characters that the shell considers to be whitespace characters.

PATH
: This variable is interpreted as a list of directories (separated by colons on a Unix system). Whenever you type the name of an executable for the shell to execute but do not include the full path of that executable, the shell will look in all of these directories *in order* to find the executable.

PS1
: This variable lists a set of codes. These codes instruct your shell about what the command-line prompt in the interactive shell should look like.

PWD
: The value of this variable is always the path of the working directory.

The absolute beauty of environment variables, as mentioned above, is that they just contain random strings of characters without an immediate meaning. The meaning of any variable is to be interpreted by whatever program or process reads the variable. So a variable can hold literally any kind of information and be used practically anywhere. For instance, consider the following example:

Environment variables are more flexible than you thought...

```
$ echo $CMD

$ CMD=ls
$ echo $CMD
ls
$ $CMD
bin  booktemp  Documents  Mail  mbox  public_html  sent
```

There's nothing wrong with setting a variable to the name of an executable, then executing that executable by calling the variable as a command.

Different kinds of environment variables

Although you use all environment variables the same way, there are a couple of different kinds of variables. In this section we discuss the differences between them and their uses.

Named variables

The simplest and most straightforward environment variable is the named variable. We saw it earlier: it's just a name with a value, which can be retrieved by prepending a '$' to the name. You create and define a named variable in one go, by typing the name, an equals sign and then something that results in a string of characters.

Earlier we saw the following, simple example:

Assigning a simple value to a variable

```
$ VAR=Hello
```

This just assigns a simple value. Once a variable has been defined, we can also redefine it:

Assigning a simple value to a variable

```
$ VAR=Goodbye
```

We aren't limited to straightforward strings either. We can just as easily assign the value of one variable to another:

Assigning a simple value to a variable

```
$ VAR=$PATH
```

We can even go all-out and combine several commands to come up with a value:

Assigning a combined value to a variable

```
$ PS1= "`whoami`@`hostname -s` `pwd` \$ "
```

In this case, we're taking the output of the three commands 'whoami', 'hostname', and 'pwd', then we add the '$' symbol, and some spacing and other formatting just to pad things out a bit. Whew. All that, just in one piece of labeled space. As you can see environment variables can hold quite a bit, including the output of entire commands.

There are usually lots of named variables defined in your environment, even if you are not aware of them. Try the 'set' command and have a look.

Positional variables

Most of the environment variables in the shell are named variables, but there are also a couple of "special" variables. Variables that you don't set, but whose values are automatically arranged and maintained by the shell. Variables which exist to help you out, to discover information about the shell or from the environment.

The most common of these are the positional or argument variables. Any command you execute in the shell (in interactive mode or in a script) can have command-line arguments. Even if the command doesn't actually use them, they can still be there. You pass command-line arguments to a command simply by typing them after the command, like so:

```
command arg0 arg1 ...
```

This is allowed for any command. Even your own shell scripts. But say that you do this (create a shell script, then execute it with arguments); how do you access the command-line arguments from your script? This is where the positional variables come in. When the shell executes a command, it automatically assigns any command-line arguments, in order, to a set of positional variables. And these variables have numbers for names: 1 through 9, accessed through $1 through $9. Well, actually zero though nine; $0 is the name of the command that was executed. For example, consider a script like this:

WithArgs.sh: A script that uses command-line arguments

```
#!/bin/sh

echo $0
echo $1
echo $2
```

And a call to this script like this:

Calling the script

Code:

```
$ WithArgs.sh Hello World
```

Output:

```
WithArgs.sh
```

```
Hello

World
```

As you can see, the shell automatically assigned the values 'Hello' and 'World' to $1 and $2 (okay, technically to the variables called 1 and 2, but it's less confusing in written text to call them $1 and $2). What happens if we call this script with more than two arguments?

Calling the script with more arguments

Code:

```
$ WithArgs.sh Hello World Mouse Cheese
```

Output:

```

WithArgs.sh
Hello
World
```

Did the mouse eat the cheese?

This is no problem whatsoever — the extra arguments get assigned to $3 and $4. But we didn't use those variables in the script, so those command-line arguments are ignored. What about the opposite case (too few arguments)?

Calling the script with too few arguments...

Code:

```
$ WithArgs.sh Hello
```

Output:

```

WithArgs.sh
Hello
```

Again, no problem. When the script accesses $2, the shell simply substitutes the value of $2 for $2. That value is nothing in this case, so we print exactly that. In this case it's not a problem, but if your script has mandatory arguments you should check whether or not they are actually there.

What about if we want 'Hello' and 'World' to be treated as one command-line argument to be passed to the script? I.e. 'Hello World' rather than 'Hello' and 'World'? We'll get deeply into that when we start talking about quoting, but for now just surround the words with single quotes:

Calling the script with multi-word arguments

Code:

```
$ WithArgs.sh 'Hello World' 'Mouse Cheese'
```

Output:

```
WithArgs.sh
Hello World
Mouse Cheese
```

There are the mouse and the cheese!

Shifting

So what happens if you have more than nine command line arguments? Then your script is too complicated. No, but seriously: then you have a little problem. It's allowed to pass more than nine arguments, but there are only nine positional variables (in Bourne Shell at least). To deal with this situation the shell includes the **shift** command:

```
shift [n]
```

*Where n is optional and a positive integer (default 1)

Shift causes the positional arguments to shift left. That is, the the value of $1 becomes the old value of $2, the value of $2 becomes the old value of $3 and so on. Using **shift**, you can access all the command-line arguments (even if there are more than nine). The optional integer argument to shift is the number of positions to shift (so you can shift as many positions in one go as you like). There are a couple of things to keep in mind though:

- No matter how often you shift, $0 always remains the original command.
- If you shift n positions, n must be lower than the number of arguments. If n is greater than the number of arguments, no shifting occurs.
- If you shift n positions, the first n arguments are lost. So make sure you have them stored elsewhere or you don't need them anymore!
- You cannot shift back to the right.

In the module on Control flow we'll see how you can go through all the arguments without knowing exactly how many there are.

Other, special variables

In addition to the positional variables the Bourne Shell includes a number of other, special variables with special information about the shell. You'll probably not use these as often, but it's good to know they're there. These variables are

$#
: The number of command-line arguments to the current command (changes after a use of the **shift** command!).

$-
: The shell options currently in effect (see the 'set' command).

$?
: The exit status of the last command executed (0 if it succeeded, non-zero if there was an error).

$$
: The process id of the current process.

$!
: The process id of the last background command.

$*
: All the command-line arguments. When quoted, expands to all command-line arguments as a single word (i.e. "$*" = "S1 $2 $3 ...").

$@
: All the command-line arguments. When quoted, expands to all command-line arguments quoted individually (i.e. "$@" = "S1" "$2" "$3" ...).

Exporting variables to a subprocess

We've mentioned it a couple of times before: Unix is a multi-user, multiprocessing operating system. And that fact is very much supported by the Bourne Shell, which allows you to start up new processes right from inside a running shell. In fact, you can even run multiple processes simultaneously next to eachother (but we'll get to that a little later). Here's a simple example of starting a subprocess:

Starting a new shell from the shell

```
$ sh
```

We've also talked about the Command Execution Environment and the Environment (the latter being a collection of variables). These environments can affect how programs run, so it's very important that they cannot inadvertently affect one another. After all, you wouldn't want the screen in your shell to go blue with yellow letters simply because somebody started Midnight Commander in another process, right?

One of the things that the shell does to avoid processes inadvertently affecting one another, is environment separation. Basically this means that whenever a new (sub)process is started, it has its own CEE and environment. Of course it would be damned inconvenient if the environment of a subprocess of your shell were *completely* empty; your subprocess wouldn't have a PATH variable or the settings you chose for the format of your prompt. On the other hand there is usually a good reason NOT to have certain variables in the environment of your subprocess, and it usually has something to do with not handing off too much environment data to a process if it doesn't need that data. This was particularly true when running copies of MS-DOS and versions of DOS under Windows. You only HAD a limited amount of environment space, so you had to use it carefully, or ask for more space on startup. These days in a UNIX environment the space issues aren't the same, but if all your existing variables ended up in the environment of your subprocess you might still adversely affect the running of the program that you started in that subprocess (there's really something to be said for keeping your environment lean and clean in the case of subprocesses).

The compromise between the two extremes that Stephen Bourne and others came up with is this: a subprocess has an environment which contains *copies* of the variables in the environment of its parent process — but only those variables that are marked to be *exported* (i.e. copied to subprocesses). In other words, you can have any variable copied into the environment of your subprocesses, but you have to let the shell know that's what you want first. Here's an example of the distinction:

Exported and non-exported variables

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
$ VAR=value
$ echo $VAR
value
$ sh
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
$ echo $VAR

$
```

In the example above, the PATH variable (which is marked for export by default) gets copied into the environment of the shell that is started within the shell. But the VAR variable is *not* marked for export, so the environment of the second shell doesn't get a copy.

In order to mark a variable for export you use the **export** command, like so:

```
export VAR0 [VAR1 VAR2 ...]
```

As you can see, you can export as many variables as you like in one go. You can also issue the **export** command without any arguments, which will print a list of variables in the environment marked for export. Here's an example of exporting a variable:

Exporting a variable

```
$ VAR=value
$ echo $VAR
value
$ sh
$ echo $VAR

$ exit #Quitting the inner shell
$ export VAR #This is back in the outer shell
$ sh
$ echo $VAR
value
```

More modern shells like Korn Shell and Bash have more extended forms of **export**. A common extension is to allow for definition and export of a variable in one single command. Another is to allow you to *remove* the export marking from a variable. However, Bourne Shell only supports exporting as explained above.

Your profile

In the previous sections we've discussed the runtime environment of every program and command you run using the shell. We've talked about the command execution environment and at some length about the piece of it simply called "the environment", which contains environment variables. We've seen that you can define your own variables and that the system usually already has quite a lot of variables to start out with.

Here's a question about those variables that the system starts out with: where do they come from? Do they descend like manna from heaven? And on a related note: what do you do if you want to create some variables automatically every time your shell starts? Or run a program every time you log in?

Those readers who have done some digging around on other operating systems will know what I'm getting at: there's usually some way of having a set of commands executed every time you log in (or every time the system starts at least). In MS-DOS for instance there is a file called autoexec.bat, which is executed every time the system boots. In older versions of MS-Windows there was system.ini. The Bourne Shell has something similar: a file in every user's home directory called *.profile*. The $HOME/.profile (HOME is a default variable whose value is your home directory) file is a shell script like any other, which is executed automatically right after you login to a new shell session. You can edit the script to have it execute any login-commands that you like.

Each specific Unix system has its own default implementation of the .profile script (including none — it's allowed not to have a .profile script). But all of them start with some variation of this:

A basic (but typical) $HOME/.profile

```
#!/bin/sh

if [ -f /etc/profile ]; then
 . /etc/profile
fi
PS1= "`whoami`@`hostname -s` `pwd` \$ "
export PS1
```

This .profile might surprise you a bit: where are all those variables that get set? Most of the variables that get set for you on a typical Unix system, also get set for all other users. In order to make that possible and easily maintainable, the common solution is to have each $HOME/.profile script start by executing another shell script: */etc/profile*. This script is a systemwide script whose contents are maintained by the system administrator (the user who logs in with username *root*). This script sets all sorts of variables and calls scripts that set even more variables and generally does everything that is necessary to provide each user with a comfortable working environment.

As you can see from the example above, you can add any personal configuration you want or need to the .profile script in your directory. The call to execute the system profile script doesn't have to be first, but you probably don't want to remove it altogether.

Multitasking and job control

With the arrival of fast computers, CPUs that can switch between multiple tasks in a very small amount of time, CPUs that can actually do multiple things at the same time and networks of multiple CPUs, having the computer perform multiple tasks at the same time has become common. Fast task switching provides the illusion that the computer really is running multiple tasks simultaneously, making it possible to effectively serve multiple users at once. And the ability to switch to a new CPU task while an old task is waiting for a peripheral device makes CPU use vastly more efficient.

In order to make use of multitasking abilities as a user, you need a command environment that supports multitasking. For example, the ability to set one program to a task, then move on and start a new program *while the old one is still running*. This kind of ability allows you as a user to do multiple things at once on the same machine, as long as those programs do not interfere. Of course, you cannot always treat each program as a "fire and forget" affair; you might have to input a password, or the program might be finished and want to tell you its results. A multitasking environment must allow you to switch between the multiple programs you have running and allow those programs to send you some sort of message if your attention is needed.

To make things a little more tangible think of something like downloading files. Usually, while you're downloading files, you want to do other stuff as well — otherwise you're going to be sitting at the keyboard twiddling your thumbs a really long time when you want to download a whole CD worth of data. So, you start up your file downloader and feed it a list of files you want to grab. Once you've entered them, you can then tell it "Go!" and it will start off by downloading the first file and continue until it finishes the last one, or until there's a problem. The smarter ones will even try to work through common problems themselves, such as files not being available. Once it starts you get the standard shell prompt back, letting you know that you can start another program.

If you want to see how far the file downloader has gotten, simply checking the files in your system against what you have on your list will tell you. But another way to notify you is via the environment. The environment can include the files that you work with, and this can help provide information about the progress of currently running programs like that file downloader. Did it download all the files? If you check the status file, you'll see that it's downloaded 65% of the files and is just working on the last three now.

Other examples of programs that don't need their hand held are programs that play music. Quite often, once you start a program that plays music tracks, you don't WANT to tell the program "Okay, now play the next track". It should be able to do that for itself, given a list of songs to play. In fact, it should not even have to hold on to the monitor; it should allow you to start running other software right after you hit the "play" button.

In this section we will explore multitasking support within the Unix shell. We will look at enabling support, at working with multiple tasks and at the utilities that a shell has available to help you.

Some terminology

Before we discuss the mechanics of multitasking in the shell, let's cover some terminology. This will help us discuss the subject clearly and you'll also know what is meant when you run across these terms elsewhere.

First of all, when we start a program running on a system in a process of its own, that process with that one running instance of the program is called a *job*. You'll also come across terms like process, task, instance or similar. But the term used in Unix shells is job. Second, the ability of the shell to influence and use multitasking (starting jobs and so on) is referred to as *job control*.

Job
: A process that is executing an instance of a computer program.

Job control
: The ability to selectively stop (suspend) the execution of jobs and continue (resume)

their execution at a later point.

Note that these terms are used this way for Unix shells. Other circumstances and other contexts might allow for different definitions. Here are some more terms you'll come across:

Job ID
: An ID (usually an integer) that uniquely identifies a job. Can be used to refer to jobs for different tools and commands.

Process ID (or PID)
: An ID (usually an integer) that uniquely identifies a process. Can be used to refer to processes for different tools and commands. Not the same as a Job ID.

Foreground job (or foreground process)
: A job that has access to the terminal (i.e. can read from the keyboard and write to the monitor).

Background job (or background process)
: A job that does not have access to the terminal (i.e. cannot read from the keyboard or write to the monitor).

Stop (or suspend)
: Stop the execution of a job and return terminal control to the shell. A stopped job is not a *terminated* job.

Terminate
: Unload a program from memory and destroy the job that was running the program.

Job control in the shell: what does it mean?

A jobs is a program you start within the shell. By default a new job will suspend the shell and take control over the input and output: every stroke you type at the keyboard will go to the job, as will every mouse movement. Nothing but the job will be able to write to the monitor. This is what we call a *foreground* job: it's in the foreground, clearly visible to you as a user and obscuring all other jobs in the system from view.

But sometimes that way of working is very clumsy and irritating. What if you start a long-running job that doesn't need your input (like a backup of your harddrive)? If this is a foreground process you have to wait until it's done before you can do anything else. In this situation you'd much rather start the program as a *background process*: a process that is running, but that doesn't listen to the input devices and doesn't write to the monitor. Unix supports them and the shell (with job control) allows you to start any job as a background job.

But what about a middle ground? Like that file downloader? You have to start it, log into a remote server, pick your files and start the download. Only after all that does it make sense for the job to be in the background. But how do you accomplish that if you've already started the program as a foreground job? Or how about this: you're busily writing a document in your favorite editor and you just want to step out to check your mail for a moment. Do you have to shut down the editor for that? And then, after you're done with your mail, restart it, re-open your file and find where you'd left off? That's inconvenient. No, a much better idea in both cases is simply to *suspend* the program: just stop it from running any further and return to the shell. Once you're back in the shell, you can start another program (mail) and then resume the suspended program (editor) when you're done with that — and return to the program exactly where you left it. Conversely, you can also decide to let the suspended process (downloader) continue running, but now in the background.

When we talk about job control in the shell, we are talking about the abilities described above: to start programs in the background, to suspend running programs and to resume suspended programs, either in the foreground or in the background.

Enabling job control

In order to do all the things we talked about in the previous section, you need two things:

- An operating system that supports job control.
- A shell that supports job control and has job control enabled.

Unix systems support multitasking and job control. Unix was designed from the ground up to support multitasking. If you come across a person claiming to be a Unix vendor but whose software doesn't support job control, call him a fraud. Then throw his install CDs away. Then throw *him* away.

Of course you've already guessed what comes next, right? I'm going to tell you Bourne Shell supports job control. And that you can rely on the same mechanisms to work in all compatible shells. Guess what: you're not correct. The original Bourne Shell has no job control support; it was a single-tasking shell. There was an extended version of the Bourne Shell though, called *jsh* (guess what the 'j' stands for...) which had job control support. To have job control in the original Bourne Shell, you had to start this extended shell in interactive mode like this:

```
jsh -i
```

Within that shell you had the job control tools we will discuss in the following sections.

Pretty much every other shell written since incorporated job control straight into the basic shell and the POSIX 1003 standard has standardized the job control utilities. So you can pretty much rely on job control being available nowadays and usually also enabled by default in interactive mode (some older shells like Korn shell had support but required you to enable that support specifically). But just in case, remember that you might have to do some extra stuff on your system to use job control. There is one gotcha though: in shell scripts, you usually include an interpreter hint that calls for a Bourne Shell (i.e. **#!/bin/sh**). Since the original Bourne Shell doesn't have job control, several modern shells turn off job control by default in non-interactive mode as a compatibility feature.

Creating a job and moving it around

We've already talked at length about how to create a foreground job: type a command or executable name at the prompt, hit enter, there's your job. Been there, done that, bought the T-shirt.

We've also already mentioned how to start a background job: by adding an ampersand at the end of the command.

Creating a background job

```
$ ls * > /dev/null &
[1] 4808
$
```

But that suddenly looks different that when we issued commands previously; there's a "[1]" and some number there. The "[1]" is the *job ID* and the number is the *process ID*. We can use these numbers to refer to the process and the job that we just created, which is useful for using tools that work with jobs. When the task finishes, you will receive a notice similar to the following:

Job done

```
[1]+  Done     ls * > /dev/null &
```

One of the tools that you use to manage jobs is the 'fg' command. This command takes a background job and places it in the foreground. For instance, consider a background job that actually takes some time to complete:

A heftier job

```
 while [ $CNT -lt 200000 ] do echo $CNT >> outp.txt; CNT=`expr $CNT + 1`; done &
```

We haven't gotten into flow control yet, but this writes 200,000 integers to a file and takes some time. It also runs in the background. Say that we start this job:

Starting the job

```
$ CNT=0
$ while [ $CNT -lt 200000 ] do echo $CNT >> outp.txt; CNT=`expr $CNT + 1`; done &
[1] 11246
```

The job is given job ID 1 and process ID 11246. Let's move the process to the foreground:

Moving the job to the front

```
$ fg %1
while [ $CNT -lt 200000 ] do
    echo $CNT >> outp.txt; CNT=`expr $CNT + 1`;
done
```

The job is now running in the foreground, as you can tell from the fact that we are not returned a prompt. Now type the **CTRL+Z** keyboard combination:

Stopping the job

```
'CTRL+Z'
[1]+  Stopped                 while [ $CNT -lt 200000 ] do
    echo $CNT >> outp.txt; CNT=`expr $CNT + 1`;
done
$
```

Did you notice the shell reports the job as stopped? Try using the 'cat' command to inspect the outp.txt file. Try it a couple of times; the contents won't change. The job is not a background job; it's not running at all! The job is suspended. Many programs recognize the **CTRL+Z** combination to suspend. And even those that don't usually have some way of suspending themselves.

Moving to the background and stopping in the background

Once a job is suspended, you can resume it either in the foreground or the background. To resume in the foreground you use the 'fg' command discussed earlier. You use 'bg' for the background:

```
bg jobId
```

To resume our long-lasting job that writes numbers, we do the following:

Resuming the job in the background

```
$ bg %1
[1]+ while [ $CNT -lt 200000 ] do
    echo $CNT >> outp.txt; CNT=`expr $CNT + 1`;
done &
$
```

The output indicates that the job is running again. In the background this time, since we are also returned a prompt.

Can we also stop a process in the background? Sure, we can move it to the foreground and hit 'CTRL+Z'. But can we also do it directly? Well, there is no utility or command to do it.

Mostly, you wouldn't *want* to do it — the whole point of putting it in the background was to let it run without bothering anybody or requiring attention. But if you really want to, you can do it like this:

```
kill -SIGSTOP jobId
```

or

```
kill -SIGSTOP processId
```

We'll get back to what this does exactly later, when we talk about signals.

Job control tools and job status

We mentioned before that the POSIX 1003.1 standard has standardized a number of the job control tools that were included for job control in the jsh shell and its successors. We've already looked at a couple of these tools; in this section we will cover the complete list.

The standard list of job control tools consists of the following:

bg
: Moves a job to the background.

fg
: Moves a job to the foreground.

jobs
: Lists the active jobs.

kill
: Terminate a job or send a signal to a process.

CTRL+C
: Terminate a job (same as 'kill' using the SIGTERM signal).

CRTL+Z
: Suspend a foreground job.

wait
: Wait for background jobs to terminate.

All of these commands can take a job specification as an argument. A job specification starts with a percent sign and can be any of the following:

%n
: A job ID (n is number).

%s
: The job whose command-line started with the string s.

%?s
: The jobs whose command-lines *contained* the string s.

%%
: The current job (i.e. the most recent one that you managed using job control).

%+
: The current job (i.e. the most recent one that you managed using job control).

%-
: The previous job.

We've already looked at 'bg', 'fg', and CTRL+Z and we'll cover 'kill' in a later section. That leaves us with 'jobs' and 'wait'. Let's start with the simplest one:

```
wait [job spec] ...
```

*Where *job spec* is a specification as listed above.

'Wait' is what you call a *synchronization mechanism*: it causes the invoking process to suspend until all background jobs terminate. Or, if you include one or more job specifications, until the jobs you list have terminated. You use 'wait' if you have fired off multiple jobs (simply to make use of a system's parallel processing capabilities) and you cannot proceed safely until they're all done.

The 'wait' command is used in quite advanced scripting. In other words, you might not use it all that often. Here's a command that you probably will use regularly though:

```
jobs [-lnprs] [job spec] ...
```

*Where

- -l lists the process IDs as well as normal output
- -n limits the output to information about jobs whose status has changed since the last status report
- -p lists only the process ID of the jobs' process group leader
- -r limits output to data on running jobs
- -s limits output to data on stopped jobs
- *job spec* is a specification as listed above

The *jobs* command reports information and status about active jobs (don't confuse active with running!). It is important to remember though, that this command reports on jobs and not processes. Since a job is local to a shell, the 'jobs' command cannot see across shells. The 'jobs' command is a primary source of information on jobs that you can apply job control to; for starters, you'll use this command to retrieve job IDs if you don't remember them. For example, consider the following:

Using 'jobs' to report on jobs

Code:

```
$ CNT0=0
$ while [ $CNT0 -lt 200000 ]; do echo $CNT0 >> outtemp0.txt; CNT0=$(expr $CNT0 + 1); done&
[1] 26859
$ CNT1=0
$ while [ $CNT1 -lt 200000 ]; do echo $CNT1 >> outtemp1.txt; CNT1=$(expr $CNT1 + 1); done&
[2] 31331
$ jobs
```

Output:

```
[1]-  Running                 while [ $CNT0 -lt 200000 ]; do
    echo $CNT0 >> outtemp0.txt; CNT0=$(expr $CNT0 + 1);
done &
[2]+  Running                 while [ $CNT1 -lt 200000 ]; do
    echo $CNT1 >> outtemp1.txt; CNT1=$(expr $CNT1 + 1);
done &
```

The 'jobs' command reports the state of active commands, including the command line and job IDs. It also indicated the current job (with a +) and the last job (with a -).

Speaking of state (which is reported by the 'jobs' command), this is a good time to talk about the different states we have. Jobs can be in any of several states, sometimes even in more than one state at the same time. The 'jobs' command reports on state directly after the job id and order. We recognize the following states:

Running
: This is where the job is doing what it's supposed to do. You probably don't need to interrupt it unless you really want to give the program your personal attention (for example, to stop the program, or to find out how far through a file download has proceeded). You'll generally find that anything in the foreground that's not waiting for your attention is in this state, unless it's been put to sleep.

Sleeping
: When programs need to retrieve input that's not yet available, there is no need for them to continue using CPU resources. As such, they will enter a sleep mode until another batch of input arrives. You will see more sleeping processes, since they are not as likely to be processing data at an exact moment of time.

Stopped
: The **stopped** state indicates that the program was stopped by the operating system. This usually occurs when the user suspends a background job (e.g. pressing CTRL-Z) or if it receives SIGSTOP. At that point, the job cannot actively consume CPU resources and aside from still being loaded in memory, won't impact the rest of the system. It will resume at the point where it left off once it receives the SIGCONT signal or is otherwise resumed from the shell. The difference between sleeping and stopped is that "sleep" is a form of waiting until a planned event happens, whereas "stop" can be user-initiated and indefinite.

Zombie

A **zombie** process appears if the parent's program terminated before the child could provide its return value to the parent. These processes will get cleaned up by the *init* process but sometimes a reboot will be required to get rid of them.

Other job control related tools

In the last section we discussed the standard facilities that are available for job control in the Unix shell. However, there are also a number of non-standard tools that you might come across. And even though the focus of this book is Bourne Shell scripting (particularly as the lingua franca of Unix shell scripting) these tools are so common that we would be remiss if we did not at least mention them.

Shell commands you might come across

In addition to the tools previously discussed, there are two shell commands that are quite common: 'stop' and 'suspend'.

```
stop job ID
```

The 'stop' command is a command that occurs in the shells of many System V-compatible Unix systems. It is used to suspend background processes — in other words, it is the equivalent of 'CTRL+Z' for background processes. It usually takes a job ID, like most of these commands. On systems that do not have a 'stop' command, you should be able to stop background processes by using the 'kill' command to send a SIGSTOP signal to the background process.

```
suspend job ID
suspend [-f]
```

The other command you might come across is the the 'suspend' command. The 'suspend' command is a little tricky though, since it doesn't always mean the same thing on all systems and all shells. There are two variations known to the authors at this time, both of which are shown above. The first, obvious one takes a job ID argument and suspends the indicated job; really it's just the same as 'CTRL+Z'.

The second variant of 'suspend' doesn't take a job ID at all, which is because it doesn't suspend any random job. Rather, it suspends the execution of the shell in which the command was issued. In this variant the -f argument indicates the shell should be suspended even if it is a login shell. To resume the shell execution, send it a SIGCONT signal using the 'kill' command.

The process snapshot utility

The last tool we will discuss is the process snapshot utility, 'ps'. This utility is not a shell tool at all, but it occurs in some variant on pretty much every system and you will want to use it often. Possibly more often even than the 'jobs' tool.

The 'ps' utility is meant to report on running processes in the system. Processes, not jobs — meaning it can see across shell instances. Here's an example of the 'ps' utility:

Using the 'ps' utility

Code:

```
$ ps x
```

Output:

```
PID    TTY     STAT  TIME    COMMAND
32094  tty5    R     3:37:21 /bin/sh
37759  tty5    S     0:00:00 /bin/ps
```

Typical output, including process state.

Typical process output includes the process ID, the ID of the terminal the process is connected to (or running on), the CPU time the process has taken and the command issued to start the process. Possibly you also get a process state. The process state is indicated by a letter code, but by-and-large the same states are reported as for job reports: **R**unning, **S**leeping, s**T**opped and **Z**ombie. Different 'ps' implementations may use different or more codes though.

The main problem with writing about 'ps' is that it is not exactly standardized, so there are different command-line option sets available. You'll have to check the documentation on your system for specific details. Some options are quite common though, so we will list them here:

-a
: List all processes except group leader processes.

-d
: List all processes except session leaders.

-e
: List all processes, without taking into account user id and other access limits.

-f
: Produce a full listing as output (i.e. all reporting options).

-g *list*
: Limit output to processes whose group leader process IDs are mentioned in *list*.

-l
: Produce a long listing.

-p *list*
    Limit output to processes whose process IDs are mentioned in *list*.
-s *list*
    Limit output to processes whose session leader process IDs are mentioned in *list*.
-t *list*
    Limit output to processes running on terminals mentioned in *list*.
-u *list*
    Limit output to processes owned by user accounts mentioned in *list*.

The 'ps' tool is useful for monitoring jobs across shell instances and for discovering process IDs for signal transmission.

Variable Expansion

In the Environment module we introduced the idea of an environment variable as a general way of storing small pieces of data. In this module we take an in-depth look at using those variables: 'variable expansion', 'parameter substitution' or just 'substitution'.

Substitution

The reason that using a variable is called substitution is that the shell literally replaces each reference to any variable with its value. This is done while evaluating the command-line, which means that the variable substitution is made *before* the command is actually executed.

The simplest way of using a variable is the way we've already seen, prepending the variable name with a '$'. So for instance:

Simple use of a variable

Code:

```
$ USER=JoeSixpack
$ echo $USER
```

Output:

```
JoeSixpack
```

The value JoeSixpack is substituted for $USER before the echo command is executed.

Of course, once the substitution is made the result is still just the text that was in the variable. The interpretation of that text is still done by whatever program is run. So for example:

Variables do not make magic

Code:

```
$ USER=JoeSixpack
$ ls $USER
```

Output:

```
ls: cannot access JoeSixpack: No such file or directory
```

Just because the text came from a variable, doesn't mean the file exists.

Basic variable expansion is already quite flexible. You can use it as described above, but you can also use variables to create longer strings. For instance, if you want to set the log directory for an application to the "log" directory in your home directory, you might fill in the setting like this:

```
$HOME/log
```

And if you're going to use that setting more often, you might want to create your own variable like this:

```
LOGDIR=$HOME/log
```

And, of course, if you want specific subdirectories for logs for different programs, then the logs for the Wyxliz application go into directory

```
$LOGDIR/Wyxliz/
```

Substitution forms

The Bourne Shell has a number of different syntaxes for variable substitution, each with its own meaning and use. In this section we examine these syntaxes.

Basic variable substitution

We've already talked at length about basic variable substitution: you define a variable, stick a '$' in front of it, the shell substitutes the value for the variable. By now you're probably bored of hearing about it.

But we've not talked about one situation that you might run into with basic variable substitution. Consider the following:

Adding some text to a variable's value

Code:

```
$ ANIMAL=duck
$ echo One $ANIMAL, two $ANIMALs
```

Output:

```
One duck, two
```

Uhhh.... we're missing something...

So what went wrong here? Well, obviously the shell substituted nothing for the ANIMAL variable, but why? Because with the extra 's' the shell thought we were asking for the non-existent ANIMALs variable. But what gives there? We've used variables in the middle of strings before (as in '/home/ANIMAL/logs'). But an 's' is not a '/': an 's' can be a valid part of a variable name, so the shell cannot tell the difference. In cases where you explicitly have to separate the variable from other text, you can use braces:

Adding some text to a variable's value, take II

Code:

```
$ ANIMAL=duck
$ echo One $ANIMAL, two ${ANIMAL}s
```

Output:

```
One duck, two ducks
```

That's better!

Both cases (with and without the braces) count as basic variable substitution and the rules are exactly the same. Just remember not to leave any spaces between the braces and the variable name.

Substitution with a default value

Since a variable can be empty, you'll often write code in your scripts to check that mandatory variables actually have a value. But in the case of optional variables it is usually more convenient not to check, but to use a default value for the case that a variable is not defined. This case is actually so common that the Bourne Shell defines a special syntax for it: the dash. Since a dash can mean other things to the shell as well, you have to combine it with braces — the final result looks like this:

```
${varname[:]-default}
```

*Where *varname* is the name of the variable

and *default* is the value used if *varname* is not defined

Again, don't leave any spaces between the braces and the rest of the text. The way to use this syntax is as follows:

Default values

Code:

```
$ THIS_ONE_SET=Hello
$ echo $THIS_ONE_SET ${THIS_ONE_NOT:-World}
```

Output:

```
Hello World
```

Compare that to this:

Default not needed

Code:

```
$ TEXT=aaaaaahhhhhhhh
$ echo Say ${TEXT:-bbbbbbbbbb}
```

Output:

```
Say aaaaaahhhhhhhh
```

Interestingly, the colon is optional; so ${VAR:-default} has the same result as ${VAR-default}.

Substitution with default assignment

As an extension to default values, there's a syntax that not only supplies a default value but assigns it to the unset variable at the same time. It looks like this:

```
${varname[:]=default}
```

*Where *varname* is the name of the variable

and *default* is the value used **and assigned** if *varname* is not defined

As usual, avoid spaces in between the braces. Here's an example that demonstrates how this syntax works:

Default value assignment

```
$ echo $NEWVAR

$ echo ${NEWVAR:=newval}
newval
$ echo $NEWVAR
newval
```

As with the default value syntax, the colon is optional.

Substitution for actual value

This substitution is sort of a quick test to see if a variable is defined (and that's usually what it's used for). It's sort of the reverse of the default value syntax and looks like this:

```
${varname[:]+substitute}
```

*Where *varname* is the name of the variable

and *substitute* is the value used if *varname* is defined

This syntax returns the substitute value if the variable **is** defined. That sounds counterintuitive at first, especially if you ask what is returned if the variable is **not** defined — and learn that the value is nothing. Here's an example:

Actual value substitution

```
$ echo ${NEWVAR:+newval}

$ NEWVAR=oldval
$ echo ${NEWVAR:+newval}
newval
```

So what could possibly be the use of this notation? Well, it's used often in scripts that have to check whether lots of variables are set or not. In this case the fact that a variable has a value means that a certain option has been activated, so you're interested in knowing that the variable *has* a value, not what that value is. It looks sort of like this (pseudocode, this won't actually work in the shell):

Default value assignment

```
if ${SPECIFIC_OPTION_VAR:+optionset} == optionset then ...
```

Of course, in this notation the colon is optional as well.

Substitution with value check

This final syntax is sort of a debug check to check whether or not a variable is set. It looks like this:

```
${varname[:]?message}
```

*Where *varname* is the name of the variable

and *message* is the printed if *varname* is not defined

With this syntax, if the variable is defined everything is okay. Otherwise, the message is printed and the command or script exits with a non-zero exit status. Or, if there is no message, the text "parameter null or not set" is printed. As usual the colon is optional and you may not have a space between the colon and the variable name.

You can use this syntax to check that the mandatory variables for your scripts have been set and to print an error message if they are not.

Default value assignment

```
$ echo ${SOMEVAR:?has not been set}
-sh: SOMEVAR: has not been set
$ echo ${SOMEVAR:?}
-sh: SOMEVAR: parameter null or not set
```

Control flow

So far we've talked about basics and theory. We've covered the different shells available and how to get shell scripts running in the Bourne Shell. We've talked about the Unix environment and we've seen that you have variables that control the environment and that you can use to store values for your own use. What we haven't done yet, though, is actually *done* anything. We haven't made the system act, jump through hoops, fetch the newspaper or do the dishes.

In this chapter it's time to get serious. In this chapter we talk programming — how to write programs that make decisions and execute commands. In this chapter we talk about control flow and command execution.

Control Flow

What is the difference between a program launcher and a command shell? Why is Bourne Shell a tool that has commanded power and respect the world over for decades and not just a stupid little tool you use to start *real* programs? Because Bourne Shell is not just an environment that launches programs: Bourne Shell is a fully programmable environment with the power of a full programming language at its command. We've already seen in Environment that Bourne Shell has variables in memory. But Bourne Shell can do more than that: it can make decisions and repeat commands. Like any real programming language, Bourne Shell has *control flow*, the ability to steer the computer.

Test: evaluating conditions

Before we can make decisions in shell scripts, we need a way of evaluating conditions. We have to be able to check the state of certain affairs so that we can base our decisions on what we find.

Strangely enough the actual shell doesn't include any mechanism for this. There is a tool for exactly this purpose called **test** (and it was literally created for use in shell scripts), but nevertheless it is not strictly part of the shell. The 'test' tool evaluates conditions and returns either **true** or **false**, depending on what it finds. It returns these values in the form of an exit status (in the $? shell variable): a zero for **true** and something else for **false**. The general form of the test command is

```
test condition
```

as in

A test for string equality

```
test "Hello World" = "Hello World"
```

This test for the equality of two strings returns an exit status of zero. There is also a shorthand notation for 'test' which is usually more readable in scripts, namely angle brackets:

```
[ condition ]
```

Note the spaces between the brackets and the actual condition – don't forget them in your own scripts. The equivalent of the example above in shorthand is

A shorter test for string equality
```
[ "Hello World" = "Hello World" ]
```

'Test' can evaluate a number of different kinds of conditions, to fit with the different kinds of tests that you're like to want to carry out in a shell script. Most specific shells have added on to the basic set of available conditions, but Bourne Shell recognizes the following:

File conditions

-b *file*
  file exists and is a block special file
-c *file*
  file exists and is a character special file
-d *file*
  file exists and is a directory
-f *file*
  file exists and is a regular data file
-g *file*
  file exists and has its set-group-id bit set
-k *file*
  file exists and has its sticky bit set
-p *file*
  file exists and is a named pipe
-r *file*
  file exists and is readable
-s *file*
  file exists and its size is greater than zero
-t [n]
  The open file descriptor with number n is a terminal device; n is optional, default 1
-u *file*
  file exists and has its set-user-id bit set
-w *file*
  file exists and is writable
-x *file*
  file exists and is executable

String conditions

-n *s*

s has non-zero length

-z *s*
: *s* has zero length

s0 = *s1*
: *s0* and *s1* are identical

s0 != *s1*
: *s0* and *s1* are different

s
: *s* is not null (often used to check that an environment variable has a value)

Integer conditions

n0 -eq *n1*
: *n0* is equal to *n1*

n0 -ge *n1*
: *n0* is greater than or equal to *n1*

n0 -gt *n1*
: *n0* is strictly greater than *n1*

n0 -le *n1*
: *n0* is less than or equal to *n1*

n0 -lt *n1*
: *n0* is strictly less than *n1*

n0 -ne *n1*
: *n0* is not equal to *n1*

Finally, conditions can be combined and grouped:

\(*B*\)
: Parentheses are used for grouping conditions (don't forget the backslashes). A grouped condition (*B*) is true if *B* is true.

! *B*
: Negation; is true if *B* is false.

B0 **-a** *B1*
: And; is true if *B0* and *B1* are both true.

B0 **-o** *B1*
: Or; is true if either *B0* or *B1* is true.

Conditional execution

Okay, so now we know how to evaluate some conditions. Let's see how make use of this ability to do some programming.

All programming languages need two things: a form of decision making or conditional execution and a form of repetition or looping. We'll get to looping later, for now let's concentrate on conditional execution. Bourne Shell supports two forms of conditional execution, the **if**-statement and the **case**-statement.

The **if**-statement is the most general of the two. It's general form is

```
if command-list
then command-list
elif command-list
then command-list
...
else command-list
fi
```

This command is to be interpreted as follows:

1. The command list following the **if** is executed.
2. If the last command returns a status zero, the command list following the first **then** is executed and the statement terminates after completion of the last command in this list.
3. If the last command returns a non-zero status, the command list following the first **elif** (if there is one) is executed.
4. If the last command returns a status zero, the command list following the next **then** is executed and the statement terminates after completion of the last command in this list.
5. If the last command returns a non-zero status, the command list following the next **elif** (if there is one) is executed and so on.
6. If no command list following the **if** or an **elif** terminates in a zero status, the command list following the else (if there is one) is executed.
7. The statement terminates. If the statement terminated without an error, the return status is zero.

It is interesting to note that the **if**-statement allows command lists everywhere, including in places where conditions are evaluated. This means that you can execute as many compound commands as you like before reaching a decision point. The only command that affects the outcome of the decision is the last one executed in the list.

In most cases though, for the sake of readability and maintainability, you will want to limit yourself to one command for a condition. In most cases this command will be a use of the 'test' tool.

Example of a simple if statement

Code:

```
if [ 1 -gt 0 ]
then
  echo YES
fi
```

Output:

```
YES
```

Example of an if statement with an else clause

Code:

```
if [ 1 -le 0 ]
then
  echo YES
else
  echo NO
fi
```

Output:

```
NO
```

Example of a full if statement with an else clause and two elifs

Code:

```
rank=captain

if [ $rank = "colonel" ]
then
  echo Hannibal Smith
elif [ $rank = "captain" ]
then
  echo Howling Mad Murdock
elif [ $rank = "lieutenant" ]
then
  echo Templeton Peck
else
  echo B.A. Baracus
fi
```

Output:

```
Howling Mad Murdock
```

The **case**-statement is sort of a special form of the **if**-statement, specialized in the kind of test demonstrated in the last example: taking a value and comparing it to a fixed set of expected values or patterns. The case statement is used very frequently to evaluate command line arguments to scripts. For example, if you write a script that uses switches to identify command line arguments, you know that there are only a limited number of legal switches. The **case**-statement is an elegant alternative to a potentially messy **if**-statement in such a case.

The general form of the case statement is

```
case value in
pattern0 ) command-list-0 ;;
pattern1 ) command-list-1 ;;
...
esac
```

The value can be any value, including an environment variable. Each pattern is a regular expression and the command list executed is the one for the first pattern that matches the value (so make sure you don't have overlapping patterns). Each command list must end with a double semicolon. The return status is zero if the statement terminates without syntax errors.

The last 'if'-example revisited

Code:

```
rank=captain

case $rank in
    colonel) echo Hannibal Smith;;
    captain) echo Howling Mad Murdock;;
    lieutenant) echo Templeton Peck;;
    sergeant) echo B.A. Baracus;;
    *) echo OOPS;;
esac
```

Output:

```
Howling Mad Murdock
```

If versus case: what is the difference?

So what exactly is the difference between the **if**- and **case**-statements? And what is the point of having two statements that are so similar? Well, the technical difference is this: the **case**-statement works off of data available to the shell (like an environment variable), whereas the **if**-statement works off the exit status of a program or command. Since fixed values and environment variables depend on the shell but the exit status is a concept general to the Unix system, this means that the **if**-statement is more general than the **case**-statement.

Let's look at a slightly larger example, just to put the two together and compare:

```
#!/bin/sh

if [ $2 ]
then
  sentence="$1 is a"
else
  echo Not enough command line arguments! >&2
  exit 1
fi

case $2 in
  fruit|veg*) sentence="$sentence vegetarian!";;
  meat) sentence="$sentence meat eater!";;
  *) sentence="${sentence}n omnivore!";;
esac

echo $sentence
```

Note that this is a shell script and that it uses positional variables to capture command-line arguments. The script starts with an **if**-statement to check that we have the right number of arguments – note the use of 'test' to see if the value of variable $2 is not null and the exit status of 'test' to determine how the **if**-statement proceeds. If there are enough arguments, we assume the first argument is a name and start building the sentence that is the result of our script. Otherwise we write an error message (to stderr, the place to write errors; read all about it in Files and streams) and exit the script with a non-zero return value. Note that this else statement has a command list with more than one command in it.

Assuming we got through the **if**-statement without trouble, we get to the **case**-statement. Here we check the value of variable $2, which should be a food preference. If that value is either fruit or something starting with veg, we add a claim to the script result that some person is a vegetarian. If the value was exactly meat, the person is a meat eater. Anything else, he is an omnivore. Note that in that last case pattern clause we have to use curly braces in the variable substitution; that's because we want to add a letter n directly onto the existing value of sentence, without a space in between.

Let's put the script in a file called 'preferences.sh' and look at the effect of some calls of this script:

Calling the script with different effects

```
$ sh preferences.sh
Not enough command line arguments!
$sh preferences.sh Joe
Not enough command line arguments!
$sh preferences.sh Joe fruit
Joe is a vegetarian!
$sh preferences.sh Joe veg
Joe is a vegetarian!
$sh preferences.sh Joe vegetables
Joe is a vegetarian!
$sh preferences.sh Joe meat
Joe is a meat eater!
sh preferences.sh Joe meat potatoes
Joe is a meat eater!
sh preferences.sh Joe potatoes
Joe is an omnivore!
```

Repetition

In addition to conditional execution mechanisms every programming language needs a means of repetition, repeated execution of a set of commands. The Bourne Shell has several mechanisms for exactly that: the **while**-statement, the **until**-statement and the **for**-statement.

The while loop

The **while**-statement is the simplest and most straightforward form of repetition statement in Bourne shell. It is also the most general. Its general form is this:

```
while command-list1
do command-list2
done
```

The **while**-statement is interpreted as follows:

1. Execute the commands in command list 1.
2. If the exit status of the last command is non-zero, the statement terminates.
3. Otherwise execute the commands in command list 2 and go back to step 1.
4. If the statement does not contain a syntax error and it ever terminates, it terminates with exit status zero.

Much like the **if**-statement, you can use a full command list to control the **while**-statement and only the last command in that list actually controls the statement. But in reality you will probably want to limit yourself to one command and, as with the **if**-statement, you will usually use the 'test' program for that command.

A while loop that prints all the values between 0 and 10

Code:

```
counter=0

while [ $counter -lt 10 ]
do
  echo $counter
  counter=`expr $counter + 1`
done
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Note the use of command substitution to increase the value of the counter variable.

The **while**-statement is commonly used to deal with situations where a script can have an indeterminate number of command-line arguments, by using the **shift** command and the special '$#' variable that indicates the number of command-line arguments:

Printing all the command-line arguments

```
#!/bin/sh

while [ $# -gt 0 ]
do
    echo $1
    shift
done
```

The until loop

The **until**-statement is also a repetition statement, but it is sort of the semantic opposite of the **while**-statement. The general form of the **until**-statement is

```
until command-list1
do command-list2
done
```

The interpretation of this statement is almost the same as that of the **while**-statement. The only difference is that the commands in command list 2 are executed as long as the last command of command list 1 returns a non-zero status. Or, to put it more simply: command list 2 is executed as long as the condition of the loop is *not* met.

Whereas **while**-statements are mostly used to establish some effect ("repeat until done"), **until**-statements are more commonly used to poll for the existence of some condition or to wait until some condition is met. For instance, assume some process is running that will write 10000 lines to a certain file. The following **until**-statement waits for the file to have grown to 10000 lines:

Waiting for myfile.txt to grow to 10000 lines

```
until [ $lines -eq 10000 ]
do
    lines=`wc -l dates | awk '{print $1}'`
    sleep 5
done
```

The for loop

In the section on Control flow, we discussed that the difference between **if** and **case** was that the first depended on command exit statuses whereas the second was closely linked to data available in the shell. That kind of pairing also exists for repetition statements: **while** and **until** use command exit statuses and **for** uses data explicitly available in the shell.

The **for**-statement loops over a fixed, finite set of values. Its general form is

```
for name in w1 w2 ...
do command-list
done
```

This statement executes the command list for each value named after the 'in'. Within the command list, the "current" value w_i is available through the variable *name*. The value list must be separated from the 'do' by a semicolon or a newline. And the command list must be separated from the 'done' by a semicolon or a newline. So, for example:

A for loop that prints some values

Code:

```
for myval in Abel Bertha Charlie Delta Easy Fox Gumbo Henry India
do
  echo $myval Company
done
```

Output:

```
Abel Company
Bertha Company
Charlie Company
Delta Company
Easy Company
Fox Company
Gumbo Company
Henry Company
India Company
```

The **for** statement is used a lot to loop over command line arguments. For that reason the shell even has a shorthand notation for this use: if you leave off the 'in' and the values part, the command assumes $* as the list of values. For example:

Using for to loop over command line arguments

Code:

```
#!/bin/sh

for arg
do
  echo $arg
done
```

Output:

```
$ sh loop_args.sh A B C D

A
B
C
D
```

This use of for is commonly combined with case to handle command line switches.

Command execution

In the last section on Control Flow we discussed the major programming constructs and control flow statements offered by the Bourne Shell. However, there are lots of other syntactic constructs in the shell that allow you to control the way commands are executed and to embed commands in other commands. In this section we discuss some of the more important ones.

Command joining

Earlier, we looked at the **if**-statement as a method of conditional execution. In addition to this expansive statement the Bourne Shell also offers a method of directly linking two commands together and making the execution of one of them conditional on the outcome (the exit status) of the other. This is useful for making quick, inline decisions on command execution. But you probably wouldn't want to use these constructs in a shell script or for longer command sequences, because they aren't the most readable.

You can join commands together using the **&&** and **||** operators. These operators (which you might recognize as borrowed from the C programming language) are *short circuiting* operators: they make the execution of the second command dependent on the exit status of the first and so can allow you to avoid unnecessary command executions.

The **&&** operator joins two commands together and only executes the second if the exit status of the first is zero (i.e. the first command "succeeds"). Consider the following example:

Attempt to create a file and delete it again *if the creation succeeds*

```
echo Hello World > tempfile.txt && rm tempfile.txt
```

In this example the deletion would be pointless if the file creation fails (because the file system is read-only, say). Using the **&&** operator prevents the deletion from being attempted if the file creation fails. A similar – and possibly more useful – example is this:

Check if a file exists and make a backup copy if it does

```
test -f my_important_file && cp my_important_file backup
```

In contrast to the **&&** operator, the **||** operator executes the second command only if the exit status of the first command is *not* zero (i.e. it fails). Consider the following example:

Make sure we do not overwrite a file; create a new file only if it doesn't exist yet

```
test -f my_file || echo Hello World > my_file
```

For both these operators the exit status of the joined commands is the exit status of the last command that actually got executed.

Command grouping

You can join multiple commands into one command list by joining them using the **;** operator, like so:

Create a directory and change into it all in one go

```
mkdir newdir;cd newdir
```

There is no conditional execution here; all commands are executed, even if one of them fails.

When joining commands into a command list, you can group the commands together for clarity and some special handling. There are two ways of grouping commands: using curly braces and using parentheses.

Grouping using curly braces is just for clarity; it doesn't add any semantics to joining using semicolons. The only differences between with braces and without are that if you use braces you *must* insert an extra semicolon after your command list and you have to remember to put spaces between the braces and your command list or the shell won't understand what you mean. Here's an example:

Create a directory and change into it all in one go, grouped with curly braces

```
{ mkdir newdir;cd newdir; }
```

The parentheses are far more interesting. When you group a command list with parentheses, it is executed... in a separate process. This means that whatever you do in the command list doesn't affect the environment in which you gave the command. Consider the example above again, with braces and parentheses:

Create a directory and change into it all in one go, grouped with curly braces

Code:

```
/home$ { mkdir newdir;cd newdir; }
```

Output:

```
/home/newdir$
```

Note that your working directory has changed

Create a directory and change into it all in one go, grouped with parentheses

Code:

```
/home$ (mkdir newdir;cd newdir)
```

Output:

```
/home$
```

Note that your working directory is still the same — but the new directory *has* been created

Here's another one:

Creating shell variables in the current and in a new environment

Code:

```
$ VAR0=A
$ (VAR1=B)
$ echo \"$VAR0\" \"$VAR1\"
```

Output:

```
"A" ""
```

VAR1 was created in a separate process with its own environment, so it doesn't exist in the original environment

Command subsitution

In the chapter on Environment we talked about variable substitution. The Bourne Shell also supports *command substitution*. This is sort of like variable substitution, but instead of a variable being replaced by its value a command is replaced by its output. We saw an example of this earlier when discussing the **while**-statement, where we assigned the outcome of an arithmetic expression evaluation to an environment variable.

Command substitution is accomplished using either of two notations. The original Bourne Shell used grave accents (`*command*`), which is still generally supported by most shells. Later on the POSIX 1003.1 standard added the **$(** *command* **)** notation. Consider the following examples:

Making a daily backup (old-skool)

```
cp myfile backup/myfile-`date`
```

Making a daily backup (POSIX 1003.1)

```
cp myfile backup/myfile-$(date)
```

Regular expressions and metacharacters

Usually, in the day-to-day tasks that you do with your shell, you will want to be explicit and exact about which files you want to operate on. After all, you want to delete a specific file and not a random one. And you want to send your network communications to the network device file and not to the keyboard.

But there are times, especially in scripting, when you will want to be able to operate on more than one file at a time. For instance, if you have written a script that makes a regular backup of all the files in your home directory whose names end in ".dat". If there are a lot of those files, or there are more being made each day with new names each time, then you do not want to have to name all those files explicitly in your backup script.

We have also seen another example of not wanting to be too explicit: in the section on the **case**-statement, there is an example where we claim that somebody is a vegetarian if he likes fruit or anything starting with "veg". We could have included all sorts of options there and been explicit (although there are an infinite number of words you can make that start with "veg"). But we used a *pattern* instead and saved ourselves a lot of time.

For exactly these cases the shell supports a (limited) form of *regular expressions*: patterns that allow you to say something like "I mean every string, every sequence of characters, that looks sort of like this". The shell allows you to use these regular expressions anywhere (although they don't always make sense — for example, it makes no sense to use a regular expression to say *where* you want to copy a file). That means in shell scripts, in the interactive shell, as part of the **case**-statement, to select files, general strings, anything.

In order to create regular expressions you use one or more *metacharacters*. Metacharacters are characters that have special meaning to the shell and are automatically recognized as part of regular expressions. The Bourne shell recognizes the following metacharacters:

*
: Matches any string.

?
: Matches any single character.

[*characters*]
: Matches any character enclosed in the angle brackets.

[!*characters*]
: Matches any character *not* enclosed in the angle brackets.

pat0|*pat1*
: Matches any string that matches *pat0* **or** *pat1* (only in **case**-statement patterns!)

Here are some examples of how you might use regular expressions in the shell:

List all files whose names end in ".dat"

```
ls *.dat
```

List all files whose names are "file-" followed by two characters followed by ".txt"

```
ls file-??.txt
```

Make a backup copy of all text files, with a datestamp

```
for i in `ls *.txt`; do cp $i backup/$i-`date +%Y%m%d`; done
```

List all files in the directories Backup0 and Backup1

```
ls Backup[01]
```

List all files in the *other* backup directories

```
ls Backup[!01]
```

Execute all shell scripts whose names start with "myscript" and end in ".sh"

```
myscript*.sh
```

Regular expressions and hidden files

When selecting files, the metacharacters match all files except files whose names start with a period ("."). Files that start with a period are either special files or are assumed to be configuration files. For that reason these files are semi-protected, in the sense that you cannot just pick them up with the metacharacters. In order to include these files when selecting with regular expressions, you must include the leading period explicitly. For example:

Lising all files whose names start with a period

Code:

```
/home$ ls .*
```

Output:

```
.
..
.profile
```

The period files in a home directory

The example above shows a listing of period files. In this example the listing includes '.profile', which is the user configuration file for the Bourne Shell. It also includes the special directories '.' (which means "the current directory") and '..' (which is the parent directory of the current directory). You can address these special directories like any other. So for instance

```
ls .
```

is the same semantically as just 'ls' and

```
cd ..
```

changes your working directory to the parent directory of the directory that was your working directory before.

Quoting

When you introduce special characters like the metacharacters discussed in the previous section, you automatically get into situations when you really don't want those special characters evaluated. For example, assume that you have a file whose name includes an asterisk ('*'). How would you address that file? For example:

Metacharacters in file names can cause problems

Code:

```
echo Test0 > asterisk*.file
echo Test1 > asteriskSTAR.file
cat asterisk*.file
```

Output:

```
Test0

Test1
```

Oops; that clearly wasn't the idea...

Clearly what is needed is a way of temporarily turning metacharacters off. The Bourne Shell built-in quoting mechanisms do exactly that. In fact, they do a lot more than that. For instance, if you have a file name with spaces in it (so that the shell cannot tell the different words in the file name belong together) the quoting mechanisms will help you deal with that problem as well.

There are three quoting mechanisms in the Bourne Shell:

\
    backslash, for single character quoting.

''
    single quotes, to quote entire strings.

""
    double quotes, to quote entire strings but still allow for some special characters.

The simplest of these is the backslash, which quotes the character that immediately follows it. So, for example:

Echo with an asterisk

Code:

```
echo *
```

Output:

```
asterisk*.file asterisking.file backup biep.txt BLAAT.txt conditional1.sh condit
ional1.sh~ conditional.sh conditional.sh~ dates error_test.sh error_test.sh~ fil
e with spaces.txt looping0.sh looping1.sh out_nok out_ok preferences.sh pre
ferences.sh~ test.txt
```

Echoing an asterisk

Code:

```
echo \*
```

Output:

```
*
```

So the backslash basically disables special character interpretation for the duration of one character. Interestingly, the newline character is also considered a special character in this context, so you can use the backslash to split commands to the interpreter over multiple lines. Like so:

A multiline command

Code:

```
echo This is a \
>very long command!
```

Output:

```
This is a very long command!
```

Note: you don't have to type the >; the shell puts it in as a hint that you're continuing on a new line.

The backslash escape also works for file names with spaces:

Difficult file to list...

Code:

```
ls file with spaces.txt
```

Output:

```
ls: cannot access file: No such file or directory
ls: cannot access with: No such file or directory
ls: cannot access spaces.txt: No such file or directory
```

Listing the file using escapes

Code:

```
ls file\ with\ spaces.txt
```

Output:

```
file with spaces.txt
```

But what if you want to pass a backslash to the shell? Well, think about it. Backslash disables interpretation of one character, so if you want to use a backslash for anything else... then '\\' will do it!

So we've seen that a backslash allows you to disable special character interpretation for a single character by quoting it. But what if you want to quote a lot of special characters all at once? As you've seen above with the filename with spaces, you *can* quote each special character separately, but that gets to be a drag really quickly. Usually it's quicker, easier and less error-prone simply to quote an entire string of characters in one go. To do exactly that you use single quotes. Two single quotes quote the entire string they surround, disabling interpretation of all special characters in that string — with the exception of the single quote (so that you can *stop* quoting as well). For example:

Quoting to use lots of asterisks

Code:

```
echo '*******'
```

Output:

```
*******
```

So let's try something. Let's assume that for some strange reason we would like to print three asterisks ("***"), then a space, then the current working directory, a space and three more asterisks. We know we can disable metacharater interpretation with single quotes so this should be no biggy, right? And to make life easy, the built-in command 'pwd' **p**rints the **w**orking **d**irectory, so this is really easy:

Printing the working directory with decorations

Code:

```
echo '*** `pwd` ***'
```

Output:

```
*** `pwd` ***
```

Uh... wait... that's not right...

So what went wrong? Well, the single quotes disable interpretation of all special characters. So the grave accents we used for the command substitution didn't work! Can we make it work a different way? Like by using the Path of Working Directory environmentvariable ($PWD)? Nope, the $-character won't work either.

This is a typical Goldilocks problem. We want to quote some special characters, but not all. We could use backslashes, but that doesn't do enough to be convenient (it's too cold). We can use single quotes, but that kills too many special characters (it's too hot). What we need is quoting that's *juuuust riiiiight*. More to the point, what we want (and more often than you think) is to disable all special character interpretation except variable and command substitution. Because this is a common desire the shell supports it through a separate quoting mechanism: the double quote. The double quote disables all special character interpretation except the grave accent (command substitution), the $ (variable substitution) and the double quote (so you can *stop* quoting). So the solution to our problem above is:

Printing the working directory with decorations, take II

Code:

```
echo "*** `pwd` ***"
```

Output:

```
*** /home/user/examples ***
```

By the way, we actually cheated a little bit above for educational purposes (hey, you try coming up with these examples); we could also have solved the problem like this:

Printing the working directory with decorations, alternative

Code:

```
echo '***' `pwd` '***'
```

Output:

```
*** /home/user/examples ***
```

Files and streams

The Unix world: one file after another

When you think of a computer and everything that goes with it, you usually come up with a mental list of all sorts of different things:

- The computer itself
- The monitor
- The keyboard
- The mouse
- Your hard drive with your files and directories on it
- The network connection leading to the Internet
- The printer
- The DVD player
- et cetera

Here's a surprise for you: Unix doesn't have any of these things. Well, almost. Unix certainly has *files*. Unix has endless reams of files. And since Unix has files, it also has a concept of "between files" (think of it this way: if your universe consists only of boxes, you automatically know about spaces where there are no boxes as well). But Unix knows nothing else than that. Everything in the whole (Unix) universe is a file.

Everything is a file. Even things that are really weird things to think of as files, are files. Your (data) files are files. Your directories are files. Your hard drive is a file. Your *keyboard*, *monitor* and *printer* are files. Yes, really: your keyboard is a read-only file of infinite size. Your monitor and printer are infinitely sized write-only files. Your network connection is a read/write file.

At this point you're probably asking: *Why?* Why would the designers of the Unix system have come up with this madness? Why is everything a file? The answer is: because if everything is a file, you can **treat** everything like a file. Or, put a different way, you can treat everything in the Unix world the same way. And, as we will see shortly, that means you can also *combine* virtually everything using file operations.

Before we move on, here's an extra level of weirdness for you: everything in Unix is a file.

Including the *processes* that run programs. Effectively this means that running programs are also files. Including the interactive shell session that you've been running to practice scripting in. Yes, really, that text screen with the blinking cursor is also a file. And we can prove it too. You might recall that in the chapter on Running Commands we mentioned you can exit the shell using the Ctrl+d key combination. Because that combination produces the Unix character for... that's right, end-of-file!

Streams: what goes between files

As we mentioned in the previous section, everything in Unix is a file -- except that which sits *between* files. Between files Unix defines a mechanism that allows data to move, bit by bit, from one file to another: the **stream**. A stream is literally what it sounds like: a little river of bits pouring from one file into another. Although actually a bridge would probably have been a better name because unlike a stream (which is a constant flow of water) the flow of bits between files need not be constant, or even used at all.

The standard streams

Within the Unix world it is a general convention that each file is connected to at least three streams (that's because that turned out to be the most useful number for those files that are processes, or running programs). There can be more and in fact each file can cause itself to be connected to any number of streams (a program can print and open a network connection, for instance). But there are three basic streams available to all files, even though they may not always be useful or used. These streams are called the "standard" streams:

Standard in (stdin)
: the standard stream for input into a file.

Standard out (stdout)
: the standard stream for output out of a file.

Standard error (stderr)
: the standard stream for error output from a file.

As you can probably tell, these streams are very geared towards those files that are actually processes of the system. In fact many programming languages (like C, C++, Java and Pascal) use exactly these conventions for their standard I/O operations. And since the Unix operating system family includes them in the core of the system definition, these streams are also central to the Bourne Shell.

Getting hold of the standard streams in your scripts

So now we know that there's a general mechanism for basic input and output in Unix; but how do you get hold of these streams in a script? What do you have to do to hook your script up to the standard out, or read from the standard in? Well, the happy answer is: nothing. Your scripts are automatically connected to the standard in, out and error stream of the process that is running them. When you read input, it automatically comes from the standard in. Your output goes straight to the standard out. And program errors go right to the standard error. In fact you've already used these streams: every example so far that has printed anything has done so to the standard output stream of your script.

And what about the shell in interactive mode? Does that use those standard streams as well? Yes, it does. In interactive mode, the standard in stream is connected to the keyboard file. And the standard output and standard error are connected to the monitor file.

Okay... But what good is it?

This discussion on files and streams has been very interesting so far and a nice insight into the depths of Unix. But what good does it do you to know all this? Ah, glad you asked!

The Bourne Shell has some built-in features that allow you to do neat tricks involving files and their streams. You see, files don't just have streams -- you can also cross-connect the streams of two files. At the end of the previous section we said that the standard input of the interactive session is connected to the keyboard file. In fact it is connected to the *standard output* stream of the keyboard file. And the standard output and error of the interactive session are connected to the *standard input* of the monitor file. So you can connect the streams of the interactive session to the streams of devices.

But wait. Do you remember the remark above that the point of Unix considering everything to be a file was that everything gets treated like a file? This is why that was important: you can connect a stream from *any* file to a stream of *any* other file. You can connect your interactive shell session to the printer or the network rather than to the monitor (or in addition to the monitor) using streams. You can run a program and have its output go directly to the printer by reconnecting the standard output stream of the program. You can connect the standard output stream of one program directly to the standard input stream of **another program** and make chains of programs. And the Bourne Shell makes it really simple to do all that.

Do you suddenly feel like you've stuck your fingers in the electrical socket? That's the feeling of the raw power of the shell flowing through your body....

Redirecting: using streams in the shell

As explained in the previous section, the shell process is connected by standard streams to (by default) the keyboard and the monitor. But very often you will want to change this linking. Connecting a file to a stream is a very common operation, so would expect it to be called something like "connecting" or "linking". But since the Bourne Shell has default connections and everything you do is always a *change* in the default connections, connecting a file to a (different) stream using the shell is actually called **redirecting**.

There are several operators built in to the Bourne Shell that relate to redirecting. The most basic and general one is the pipe operator, which we will examine in some detail further on. The others are related to redirecting to file.

Redirecting to file

As we explained (or rather: hinted at) in the previous section, one of the enormously powerful features of the Bourne Shell on top of a Unix operating system is the ability to chain programs together. Execute a program, have it produce output, then automatically send that output to another program as input. The possible combinations are endless, as is the power of what you can achieve.

One of the most common places where you might want to send a program's output is to a file in the file system. And this time by file we mean a regular, classic data file and not a Unix "everything is a file including your hardware" file. In order to achieve this you can imagine that we can use the chaining mechanism described above: let a program generate output through the standard output stream, then connect that stream (i.e. *redirect the output*) to the standard input stream of a program that creates a data file in the file system. And this would absolutely work. However, redirecting to a data file is such a common operation that you don't need a separate end-of-chain program for it. Redirecting to file is built straight into the Bourne Shell, through the following operators:

process > *data file*
: redirect the output of *process* to the data file; create the file if necessary, overwrite its existing contents otherwise.

process >> *data file*
: redirect the output of *process* to the data file; create the file if necessary, append to its existing contents otherwise.

process < *data file*
: read the contents of the data file and redirect that contents to *process* as input.

Redirecting output

Let's take a closer look at these operators through some examples. Take the simple Bourne shell script below called 'hello.sh':

A simple shell script that generates some output

```
#!/bin/sh
echo Hello
```

This code may be run in any of the ways described in the chapter Running Commands. When we run the script, it simply outputs the string "Hello" to the screen and then returns us to our prompt. But let's say we want to redirect the output to a file instead. We can use the redirect operators to do that easily:

Redirecting the output to a data file

```
$ hello.sh > myfile.txt
$
```

This time, we don't see the string 'Hello' on the screen. Where's it gone? Well, exactly where we wanted it to: into the (new) data file called 'myfile.txt'. Let's examine this file using the 'cat' command:

Examining the results of redirecting some output

```
$ cat myfile.txt
Hello
$
```

Let's run the program again, this time using the '>>' operator instead, and then examine 'myfile.txt' again using the 'cat' command:

Redirecting using the append redirect

```
$ hello.sh >> myfile.txt
$ cat myfile.txt
Hello
Hello
$
```

You can see that 'myfile.txt' now consists of two lines — the output has been added to the end of the file (or concatenated); this is due to the use of the '>>' operator. If we run the script again, this time with the single greater-than operator, we get:

Redirecting using the overwrite redirect

```
$ hello.sh > myfile.txt
$ cat myfile.txt
Hello
$
```

Just one 'Hello' again, because the '>' will always overwrite the contents of an existing file if there is one.

Redirecting input

Okay, so it's clear we can redirect output to a data file. But what about reading from a data file? That's also pretty common. The Bourne Shell helps us here as well: the entire process of reading a file and pumping its data into a stream is captured by the '<' operator.

By default 'stdin' is fed from your keyboard; run the 'cat' command without any arguments and it will just sit there, waiting for you type something:

cat ???

```
$ cat


I can type all day here, and I never seem to get my prompt back from
this stupid machine.

I have even pressed RETURN a few times !!!
.....etc....etc
```

In fact 'cat' will sit there all day until you type a 'Ctrl+D' (the 'End of File Character' or 'EOF' for short). To redirect our standard input from somewhere else use the '<' (less-than operator):

Redirecting into the standard input

```
$ cat < myfile.txt
Hello
$
```

So 'cat' will now read from the text file 'myfile.txt'; the 'EOF' character is also generated at the end of file, so 'cat' will exit as before.

Note that we previously used 'cat' in this format:

```
$ cat myfile.txt
```

Which is functionally identical to

```
$ cat < myfile.txt
```

However, these are two fundamentally different mechanisms: one uses an argument to the command, the other is more general and redirects 'stdin' – which is what we're concerned with here. It's more convenient to use 'cat' with a filename as argument, which is why the inventors of 'cat' put this in. However, not all programs and scripts are going to take arguments so this is just an easy example.

Combining file redirects

It's possible to redirect 'stdin' and 'stdout' in one line:

Redirecting input to and output from cat at the same time

```
$ cat < myfile.txt > mynewfile.txt
```

The command above will copy the contents of 'myfile.txt' to 'mynewfile.txt' (and will overwrite any previous contents of 'mynewfile.txt'). Once again this is just a convenient example as we normally would have achieved this effect using 'cp myfile.txt mynewfile.txt'.

Redirecting standard error (and other streams)

So far we have looked at redirecting the "normal" standard streams associated with files, i.e. the files that you use if everything goes correctly and as planned. But what about that other stream? The one meant for errors? How do we go about redirecting that? For example, if we wanted to redirect error data into a log file.

As an example, consider the ls command. If you run the command 'ls myfile.txt', it simply lists the filename 'myfile.txt' &ndash if that file exists. If the file 'myfile.txt' does NOT exist, 'ls' will return an error to the 'stderr' stream, which by default in Bourne Shell is also connected to your monitor.

So, lets run 'ls' a couple of times, first on a file which does exist and then on one that doesn't:

Listing an existing file

Code:

```
$ ls myfile.txt
```

Output:

```
myfile.txt $
```

and then:

Listing a non-existent file

Code:

```
$ ls nosuchfile.txt
```

Output:

```
ls: no such file or directory
$
```

And again, this time with 'stdout' redirected only:

Trying to redirect...

Code:

```
$ ls nosuchfile.txt > logfile.txt
```

Output:

```
ls: no such file or directory
$
```

We still see the error message; 'logfile.txt' will be created but will be empty. This is because we have now redirected the stdout stream, while the error message was written to the error stream. So how do we tell the shell that we want to redirect the error stream?

In order to understand the answer, we have to cover a little more theory about Unix files and streams. You see, deep down the reason that we can redirect stdin and stdout with simple operators is that redirecting those streams is so common that the shell lets us use a shorthand notation for those streams. But actually, to be completely correct, we should have told the shell in every case *which* stream we wanted to redirect. In general you see, the shell cannot know: there could be tons of streams connected to any file. And in order to distinguish one from the other each stream connected to a file has a number associated with it: by convention 0 is the standard in, 1 is the standard out, 2 is standard error and any other streams have numbers counting on from there. To redirect any particular stream you prepend the redirect operator with the stream number (called the *file descriptor*. So to redirect the error message in our example, we prepend the redirect operator with a 2, for the stderr stream:

Redirecting the stderr stream

Code:

```
$ ls nosuchfile.txt 2> logfile.txt
```

Output:

```
$
```

No output to the screen, but if we examine 'logfile.txt':

Checking the logfile

Code:

```
$ cat logfile.txt
```

Output:

```
ls: no such file or directory
$
```

As we mentioned before, the operator without a number is a shorthand notation. In other words, this:

```
$ cat < inputfile.txt > outputfile.txt
```

is actually short for

```
$ cat 0< inputfile.txt 1> outputfile.txt
```

We can also redirect both 'stdout' and 'stderr' independently like this:

```
$ ls nosuchfile.txt > stdio.txt 2>logfile.txt
$
```

'stdio.txt' will be blank , 'logfile.txt' will contain the error as before.

If we want to redirect stdout and stderr to the same file, we can use the file descriptor as well:

```
$ ls nosuchfile.txt > alloutput.txt 2>&1
```

Here '2>&1' means something like 'redirect stderr to the same file stdout has been redirected to'. Be careful with the ordering! If you do it this way:

```
$ ls nosuchfile.txt 2>&1 > alloutput.txt
```

you will redirect stderr to the file that stdout points to, then send stdout somewhere else — and both streams will end up being redirected to different locations.

Special files

We said earlier that the redirect operators discussed so far all redirect to data files. While this is technically true, Unix magic still means that there's more to it than just that. You see, the Unix file system tends to contain a number of special files called "devices", by convention collected in the /dev directory. These device files include the files that represent your hard drive, DVD player, USB stick and so on. They also include some special files, like /dev/null (also known as the bit bucket; anything you write to this file is discarded). You can redirect to device files as well as to regular data files. Be careful here; you really don't want to redirect raw text data to the boot sector of your hard drive (and you can!). But if you know what you're doing, you can use the device files by redirecting to them (this is how DVDs are burned in Linux, for instance).

As an example of how you might actually use a device file, in the 'Solaris' flavour of Unix the loudspeaker and its microphone can be accessed by the file '/dev/audio'. So:

```
# cat /tmp/audio.au > /dev/audio
```

Will play a sound, whereas:

```
# cat < /dev/audio > /tmp/mysound.au
```

Will record a sound.(you will need to CTRL-C this to finish...)

This is fun:

```
# cat < /dev/audio > /dev/audio
```

Now wave the microphone around whilst shouting - Jimmy Hendrix style feedback. Great stuff. You will probably need to be logged in as 'root' to try this by the way.

Some redirect warnings

The astute reader will have noticed one or two things in the discussion above. First of all, a file can have more than just the standard streams associated with it. Is it legal to redirect those? Is it even *possible*? The answer is, technically, yes. You can redirect stream 4 or 5 of a file (if they exist). Don't try it though. If there's more than a few streams in any direction, you won't know which stream you're redirecting. Plus, if a program needs more than the standard streams it's a good bet that program also needs its extra streams going to a specific location.

Second, you might have noticed that file descriptor 0 is, by convention, the standard input stream. Does that mean you can redirect a program's standard input **away** from the program? Could you do the following?

```
$ cat 0> somewhere_else
```

The answer is, yes you can. And yes, things will break if you do.

Pipes, Tees and Named Pipes

So, after all this talk about redirecting to file, we finally get to it: general redirecting by cross-connecting streams. The most general form of redirecting and the most powerful one to boot. It's called a pipe and is performed using the pipe operator '|'. Pipes allow you to join two processes together through a "pipeline", which directly connects the stdout of one file to the stdin of another.

As an example let's consider the 'grep' command which returns a matching string, given a keyword and some text to search. And let's also use the ps command, which lists running processes on the machine. If you give the command

```
$ ps -eaf
```

it will generally list pagefuls of running processes on your machine, which you would have to sift through manually to find what you want. Let's say you are looking for a process which you know contains the word 'oracle'; use the output of 'ps' to pipe into grep, which will only return the matching lines:

```
$ ps -eaf | grep oracle
```

Now you will only get back the lines you need. What happens if there's still loads of these ? No problem, pipe the output to the command 'more' (or 'pg'), which will pause your screen if it fills up:

```
$ ps -ef | grep oracle | more
```

What about if you want to kill all those processes? You need the 'kill' program, plus the process number for each process (the second column returned by the ps command). Easy:

```
$ ps -ef | grep oracle | awk '{print $2}' | xargs kill -9
```

In this command, 'ps' lists the processes and 'grep' narrows the results down to oracle. The 'awk' tool pulls out the second column of each line. And 'xargs' feeds each line, one at a time, to 'kill' as a command line argument.

Pipes can be used to link as many programs as you wish within reasonable limits (and we don't know what these limits are!)

Don't forget you can still use the redirectors in combination:

```
$ ps -ef | grep oracle > /tmp/myprocesses.txt
```

There is another useful mechanism that can be used with pipes: the 'tee'. To understand tee, imagine a pipe shaped like a 'T' - one input, two outputs:

```
$ ps -ef | grep oracle | tee /tmp/myprocesses.txt
```

The 'tee' will copy whatever is given to its stdin and redirect this to the argument given (a file); it will also then send a further copy to its stdout - which means you can effectively intercept the pipe, take a copy at this stage, and carry on piping up other commands; useful maybe for outputting to a logfile, and copying to the screen.

A note on piped commands: piped processes run in parallel on the Unix environment. Sometimes one process will be blocked, waiting for input from another process. But each process in a pipeline is, in principle, running simultaneously with all the others.

Named pipes

There is a variation on the in-line pipe which we have been discussing called the 'named pipe'. A named pipe is actually a file with its own 'stdin' and 'stdout' - which you attach processes to. This is useful for allowing programs to talk to each other, especially when you don't know exactly when one program will try and talk to the other (waiting for a backup to finish etc) and when you don't want to write a complicated network-based listener or do a clumsy polling loop.

To create a 'named pipe', you use the 'mkfifo' command (fifo=first in, first out; so data is read out in the same order as it is written into).

```
$ mkfifo mypipe
$
```

This creates a named pipe called 'mypipe'; next we can start using it.

This test is best run with two terminals logged in:

1. From 'terminal a'

```
$ cat < mypipe
```

The 'cat' will sit there waiting for an input.

2. From 'terminal b'

```
$ cat myfile.txt > mypipe
 $
```

This should finish immediately. Flick back to 'terminal a'; this will now have read from the pipe and received an 'EOF', and you will see the data on the screen; the command will have finished, and you are back at the command prompt.

Now try the other way round:

1. From terminal 'b'

```
$ cat myfile.txt > mypipe
```

This will now sit there, as there isn't another process on the other end to 'drain' the pipe - it's blocked.

2. From terminal 'a'

```
$ cat < mypipe
```

As before, both processes will now finish, the output showing on terminal 'a'.

Here documents

So far we have looked at redirecting from and to data files and cross-connecting data streams. All of these shell mechanisms are based on having a "physical" source for data — a process or a data file. Sometimes though, you want to feed some data into a target without having a source for it. In these cases you can use an "on the fly" document called a *here document*. A here document means that you open a virtual text document (in memory), type into it as usual, close it and then treat it like any normal file.

Creating a here document is done using a variation on the input redirect operator: the '<<' operator. Like the input redirect operator, the here document operator takes an argument. For the input redirect operator this operand is the name of the file to be streamed in. For the here document operator it is the string that will terminate the here document. So using the here document operator looks like this:

```
target << terminator string
here document contents
terminator string
```

For example:

Using a here document

Code:

```
cat << %%
> This is a test.
> This test uses a here document.
> Hello world.
> This here document will end upon the occurrence of the string "%%" on a separate line.
> So this document is still open now.
> But now it will end....
> %%
```

Output:

```
This is a test.
This test uses a here document.
Hello world.
This here document will end upon the occurrence of the string "%%" on a separate line.
So this document is still open now.
But now it will end....
```

When using here documents in combination with variable or command substitution, it is important to realize that substitutions are carried out *before* the here document is passed on. So for example:

Using a here document with substitutions

Code:

```
$ COMMAND=cat
$ PARAM='Hello World!!'
$ $COMMAND <<%
> `echo $PARAM`
> %
```

Output:

```
Hello World!!
```

Modularization

If you've ever done any programming in a different environment than the shell, you're probably familiar with the following scenario: you're writing your program, happily typing away at the keyboard, until you notice that

- you have to repeat some code you typed earlier because your program has to perform exactly the same actions in two different locations; or
- your program is just too *long* to understand anymore.

In other words, you've reached the point where it becomes necessary to divide your program up into modules that can be run as separate subprograms and called as often as you like. Working in the Bourne Shell is no different than working in any other language in this respect. Sooner or later you're going to find yourself writing a shell script that's just too long to be practical anymore. And the time will have come to divide your script up into modules.

Named functions

Of course, the easy and obvious way to divide a script into modules is just to create a couple of different shell scripts — just a few separate text files with executable permissions. But using separate files isn't always the most practical solution either. Spreading your script over multiple files can make it hard to maintain. Especially if you end up with shell scripts that aren't really meaningful unless they are called specifically from one other, particular shell script.

Especially for this situation the Bourne Shell includes the concept of a *named function*: the possibility to associate a name with a command list and execute the command list by using the name as a command. This is what it looks like:

```
name () command group
```

*Where *name* is a text string

and *command group* is any grouped command list (either with curly braces or parentheses)

This functionality is available throughout the shell and is useful in several situations. First of all, you can use it to break a long shell script up into multiple modules. But second, you can use it to define your own little macros in your own environment that you don't want to create a full script for. Many modern shells include a built-in command for this called *alias*, but old-fashioned shells like the original Bourne Shell did not; you can use named functions to accomplish the same result.

Creating a named function

Functions with a simple command group

Let's start off simply by creating a function that prints "Hello World!!". And let's call it "hw". This is what it looks like:

Hello world as a named function

```
hw() {
>  echo 'Hello World!!';
>}
```

We can use exactly the same code in a shell script or in the interactive shell — the example above is from the interactive shell. There are several things to notice about this example. First of all, we didn't need a separate keyword to define a function, just the parentheses did it. To the shell, function definitions are like extended variable definitions. They're part of the environment; you set them just by defining a name and a meaning.

The second thing to note is that, once you're past the parentheses, all the normal rules hold for the command group. In our case we used a command group with braces, so we needed the semicolon after the echo command. The string we want to print contains exclamation points, so we have to quote it (as usual). And we were allowed to break the command group across multiple lines, even in interactive mode, just like normal.

Here's how you use the new function:

Calling our function

Code:

```
$ hw
```

Output:

```
Hello World!!
```

Functions that execute in a separate process

The definition of a function takes a command group. *Any* command group. Including the command group with parentheses rather than braces. So if we want, we can define a function that runs as a subprocess in its own environment as well. Here's hello world again, in a subprocess:

Hello world as a named function

```
hw() ( echo 'Hello World!!' )
```

It's all on one line this time to keep it short, but the same rules apply as before. And of course the same environment rules apply as well, so any variables defined in the function will not be available anymore once the function ends.

Functions with parameters

If you've done any programming in a different programming language you know that the most useful functions are those that take parameters. In other words, ones that don't always rigidly do the same thing but can be influenced by values passed in when the function is called. So here's an interesting question: can we pass parameters to a function? Can we create a definition like

```
functionWithParams (ARG0, ARG1) { do something with ARG0 and ARG1 }
```

And then make a call like 'functionWithParams(Hello, World)'? Well, the answer is simple: no. The parenthese are just there as a flag for the shell to let it know that the previous name is the name of a function rather than a variable and there is no room for parameters.

Or actually, it's more a case of the above being the simple answer rather than the answer being simple. You see, when you execute a function you are executing a command. To the shell there's really very little difference between executing a named function and executing 'ls'. It's a command like any other. And it may not be able to have parameters, but like any other command it can certainly have *command line arguments*. So we may not be able to define a function with parameters like above, but we can certainly do this:

Functions with command-line arguments

Code:

```
$ repeatOne () { echo $1; }
$ repeatOne 'Hello World!'
```

Output:

```
Hello World!
```

And you can use any other variable from the environment as well. Of course, that's a nice trick for when you're calling a function from the command line in the interactive shell. But what about in a shell script? The positional variables for command-line arguments are already taken by the arguments to the shell script, right? Ah, but wait! Each command executed in the shell (no matter how it was executed) has its **own** set of command-line arguments! So thereś no interference and you can use the same mechanism. For example, if we define a script like this:

function.sh: A function in a shell script

```
#!/bin/sh

myFunction() {
  echo $1
}

echo $1
myFunction
myFunction "Hello World"
echo $1
```

Then it executes exactly the way we want:

Executing the function.sh script

Code:

```
$ . function.sh 'Goodbye World!!'
```

Output:

```
Goodbye World!


Hello World

Goodbye World!
```

Functions in the environment

We've mentioned it before, but let's delve a little deeper into it now: what are functions exactly? We've hinted that they're an alias for a command list or a macro and that they're part of the environment. But what *is* a function exactly?

A function, as far as the shell is concerned, is just a very verbose variable definition. And that's really all it is: a name (a text string) that is associated with a value (some more text) and can be replaced by that value when the name is used. Just like a shell variable. And we can prove it too: just define a function in the interactive shell, then give the 'set' command (to list all the variable definitions in your current environment). Your function will be in the list.

Because functions are really a special kind of shell variable definition, they behave exactly the same way "normal" variables do:

- Functions are defined by listing their name, a definition operator and then the value of the function. Functions use a different definition operator though: '()' instead of '='. This tells the shell to add some special considerations to the function (like not needing the '$' character when using the function).
- Functions are part of the environment. That means that when commands are issued from the shell, functions are also copied into the copy of the environment that is given to the issued command.
- Functions can also be passed to new subprocesses if they are marked for export, using the 'export' command. Some shells will require a special command-line argument to 'export' for functions (bash, for instance, requires you to do an 'export -f' to export functions).
- You can drop function definitions by using the 'unset' command.

Of course, when you use them functions behave just like commands (they are expanded into a command list, after all). We've already seen that you can use command-line arguments with functions and the positional variables to match. But you can also redirect input and output to and from commands and pipe commands together as well.

Debugging and signal handling

In the previous sections we've told you all about the Bourne Shell and how to write scripts using the shell's language. We've been careful to include all the details we could think of so you will be able to write the best scripts you can. But no matter how carefully you've paid attention and no matter how carefully you write your scripts, the time will come to pass when something you've written simply will not work — no matter how sure you are it should. So how do you proceed from here?

In this module we cover the tools the Bourne Shell provides to deal with the unexpected. Unexpected behavior of your script (for which you must debug the script) and unexpected behavior *around* your script (caused by signals being delivered to your script by the operating system).

Debugging Flags

So here you are, in the middle of the night having just finished a long and complicated shell script, just poured your heart and soul into it for three days straight while living on nothing but coffee, cola and pizza... and it just won't work. Somewhere in there is a bug that is just eluding you. Something is going wrong, some unexpected behavior has popped up or something else is driving you crazy. So how are you going to debug this script? Sure, you can pump it full of 'echo' commands and debug that way, but isn't there an easier way?

Generally speaking the most insightful way to debug any program is to follow the execution of the program along statement by statement to see what the program is doing exactly. The most advanced form of this (offered by modern IDEs) allows you to trace into a program by stopping the execution at a certain point and examining its internal state. The Bourne Shell is, unfortunately, not that advanced. But it does offer the next best thing: command tracing. The shell can print each and every command as it is being executed.

The tracing functionality (there are two of them) is activated using either the 'set' command or by passing parameters directly to the shell executable when it is called. In either case you can use the *-x* parameter, the *-v* parameter or both.

-v
: Turns on verbose mode; each command is printed by the shell as it is read.

-x
: This turns on command tracing; every command is printed by the shell as it is executed.

Debugging

Let's consider the following script:

divider.sh: Script with a potential error

```
#!/bin/sh

DIVISOR=${1:-0}
echo $DIVISOR
expr 12 / $DIVISOR
```

Let's execute this script and not pass in a command-line argument (so we use the default value 0 for the DIVISOR variable):

Running the script

Code:

```
$ sh divider.sh
```

Output:

```
0
expr: division by zero
```

Of course it's not too hard to figure out what went wrong in this case, but let's take a closer look anyway. Let's see what the shell executed, using the **-x** parameter:

Running the script with tracing on

Code:

```
$ sh -x divider.sh
```

Output:

```
+ DIVISOR=0

+ echo 0
0
+ expr 12 / 0

expr: division by zero
```

So indeed, clearly the shell tried to have a division by zero evaluated. Just in case we're confused about where the zero came from, let's see which commands the shell actually read:

Running the script in verbose mode

Code:

```
$ sh -v divider.sh
```

Output:

```
#!/bin/sh


DIVISOR=${1:-0}
```

```
echo $DIVISOR
0
expr 12 / $DIVISOR

expr: division by zero
```

So obviously, the script read a command with a variable substitution that didn't work out very well. If we combine these two parameters the resulting output tells the whole, sad story:

Running the script with maximum debugging

Code:

```
$ sh -xv divider.sh
```

Output:

```
#!/bin/sh


DIVISOR=${1:-0}
+ DIVISOR=0
echo $DIVISOR
+ echo 0
0
expr 12 / $DIVISOR
+ expr 12 / 0

expr: division by zero
```

There is another parameter that you can use to debug your script, the **-n** parameter. This causes the shell to read the commands but not execute them. You can use this parameter to do a syntax checking run of your script.

Places to put your parameters

As you saw in the previous section, we used the shell parameters by passing them in as command-line parameters to the shell executable. But couldn't we have put the parameters inside the script itself? After all, there is an interpreter hint in there... And surely enough, we can do exactly that. Let's modify the script a little and try it.

The same script, but now with parameters to the interpreter hint

```
#!/bin/sh -xv

DIVISOR=${1:-0}
echo $DIVISOR
expr 12 / $DIVISOR
```

Running the script

Code:

```
$ chmod +x divider.sh
$ ./divider.sh
```

Output:

```
#!/bin/sh


DIVISOR=${1:-0}
+ DIVISOR=0
echo $DIVISOR
+ echo 0
0
expr 12 / $DIVISOR
+ expr 12 / 0
expr: division by zero
```

Works like a charm!

So there's no problem there. But there is a little gotcha. Let's try running the script again:

Running the script again

Code:

```
$ sh divider.sh
```

Output:

```
0

expr: division by zero
```

Where did the debugging go?

So what happened to the debugging that time? Well, you have remember that the interpreter hint is used when you try to execute the script as an executable in its own right. But in the last example, we weren't doing that. In the last example we called the shell ourselves and passed it the script as a parameter. So the shell executed without any debugging activated. It would have worked if we'd done a "sh -xv divider.sh" though.

What about sourcing the script (i.e. using the dot notation)?

Running the script again

Code:

```
$ . divider.sh
```

Output:

```
0

expr: division by zero
```

No debugging there either...

This time the script was executed by the same shell process that is running the interactive shell for us. And the same principle applies: no debugging there either. Because the interactive shell was not started with debugging flags. But we can fix that as well; this is where the 'set' command comes in:

Running the script again

Code:

```
$ set -xv
$ . divider.sh
```

Output:

```
. divider.sh

+ . divider.sh
#!/bin/sh -vx

DIVISOR=${1:-0}
++ DIVISOR=0
echo $DIVISOR
++ echo 0
0
expr 12 / $DIVISOR
++ expr 12 / 0
expr: division by zero
```

And there we are, full tracing.

And now we have debugging active in the interactive shell and we get a full trace of the script. In fact, we even get a trace of the interactive shell *calling* the script! But now what happens if we start a new shell process with debugging on in the interactive shell? Does it carry over?

Running the script again

Code:

```
$ sh divider.sh
```

Output:

```
sh divider.sh

+ sh divider.sh
0
expr: division by zero
```

Not quite...

Well, we certainly get a trace of the script being called, but no trace of the script itself. The moral of the story is: when debugging, make sure you know which shell you're activating the trace in.

By the way, to turn tracing in the interactive shell off again you can either do a 'set +xv' or simply a 'set -'.

Breaking out of a script

When writing or debugging a shell script it is sometimes useful to exit out (to stop the execution of the script) at certain points. You use the 'exit' built-in command to do this. The command looks simply like this:

```
exit [n]
```

* Where n (optional) is the exit status of the script.

If you leave off the optional exit status, the exit status of he script will be the exit status of the last command that executed before the call to 'exit'.

For example:

Exiting from a script

```
#!/bin/sh -x
echo hello
exit 1
```

If you run this script and then test the output status, you will see (using the "$?" built-in variable):

Checking the exit status

Code:

```
echo $?
```

Output:

```
1
```

There's one thing to look out for when using 'exit': 'exit' actually terminates the executing process. So if you're executing an executable script with an interpreter hint or have called the shell explicitly and passed in your script as an argument that is fine. But if you've sourced the script (used the dot-notation), then your script is being run by the process running your interactive shell. So you may inadvertently terminate your shell session and log yourself out by using the 'exit' command!

There's a variation on 'exit' meant specifically for blocks of code that are not processes of their own. This is the 'return' command, which is very similar to the 'exit' command:

```
return [n]
```

* Where n (optional) is the exit status of the block.

Return has exactly the same semantics as 'exit', but is primarily intended for use in shell functions (it makes a function return without terminating the script). Here's an example:

exit_and_return.sh: A script with a function and an explicit return

```
#!/bin/sh

sayHello() {
  echo 'Hi there!!'
  return 2
}

echo 'Hello World!!'
sayHello
echo $?
echo 'Goodbye!!'
exit
```

If we run this script, we see the following:

Running the script

Code:

```
./exit_and_return.sh
```

Output:

```
Hello World!!

Hi there!!
2

Goodbye!!
```

The function returned with a testable exit status of 2. The overall exit status of the script is zero though, since the last command executed by the script ('echo Goodbye!!') exited without errors.

You can also use a 'return' statement to exit a shell script that you executed by sourcing it (the script will be run by the process that runs the interactive shell, so that's not a subprocess). But it's usually not a good idea, since this will limit your script to being sourced: if you try to run it any other way, the fact that you used a 'return' statement alone will cause an error.

Signal trapping

A syntax, command error or call to 'exit' is not the only thing that can stop your script from executing. The process that runs your script might also suddenly receive a signal from the operating system. Signals are a simple form of event notification: think of a signal as a little light suddenly popping on in your room, to let you know that somebody outside the room wants your attention. Only it's not just one light. The Unix system usually allows for lots of different signals so it's more like having a wall full of little lamps, each of which could suddenly start blinking.

On a single-process operating system like MS-DOS, life was simple. The environment was single-process, meaning your code (once running) had complete machine control. Any signal arriving was always a hardware interrupt (e.g. the computer signalling that the floppy disk was ready to read) and you could safely ignore all those signals if you didn't need external hardware; either it was some device event you weren't interested in, or something was really wrong — in which case the computer was crashing anyway and there was nothing you could do.

On a Unix system, life is not so easy. On Unix signals can come from all over the place (including other programs). And you never have complete control of the system either. A signal may be a hardware interrupt, or another program signalling, or the user who got fed up with waiting, logged in to a second shell session and is now ordering your process to die. On the bright side, life is still not too complicated. Most Unix systems (and certainly the Bourne Shell) come with default handling for most signals. Usually you can still safely ignore signals and let the shell or the OS deal with them. In fact, if the signal in question is number 9 (loosely translated: **KILL!! KILL!! DIE!! DIE, RIGHT NOW!!**), you probably *should* ignore it and let the OS kill your process.

But sometimes you just have to do your own signal handling. That might be because you've been working with files and want to do some cleanup before your process dies. Or because the signal is part of your multi-process program design (e.g. listening for signal 16, which is "user-defined signal 1"). Which is why the Bourne Shell gives us the 'trap' command.

Trap

The trap command is actually quite simple (especially if you've ever done event-driven programming of any kind). Essentially the trap command says "if one of the following signals is received by this process, do this". It looks like this:

```
trap [command string] signal0 [signal1] ...
```

*Where *command string* is a string containing the commands to execute if a signal is trapped

and *signal$_n$* is a signal to be trapped.

For example, to trap user-defined signal 1 (commonly referred to as SIGUSR1) and print "Hello World" whenever it comes along, you would do this:

Trapping SIGUSR1

```
$ trap "echo Hello World" 16
```

Most Unix systems also allow you to use symbolic names (we'll get back to these a little later). So you can probably also do this:

Trapping SIGUSR1 (little easier)

```
$ trap "echo Hello World" SIGUSR1
```

And if you can do that, you can usually also do this:

Trapping SIGUSR1 (even easier)

```
$ trap "echo Hello World" USR1
```

The command string passed to 'trap' is a string that contains a command list. It's not treated as a command list though; it's just a string and it isn't interpreted until a signal is caught. The command string can be any of the following:

A string
: A string containing a command list. Any and all commands are allowed and you can use multiple commands separated by semicolons as well (i.e. a command list).

''
: The empty string. Actually this is the same as the previous case, since this is the empty command string. This causes the shell to execute nothing when a signal is trapped — in other words, to ignore the signal.

: Nothing, the null string. This resets the signal handling to the default signal action (which is usually "kill process").

Following the command list you can list as many signals as you want to be associated with that command list. The traps that you set up in this manner are valid for every command that *follows* the 'trap' command.

Right about now it's probably a good idea to look at an example to clear things up a bit. You can use 'trap' anywhere (as usual) including the interactive shell. But most of the time you will want to introduce traps into a script rather than into your interactive shell process. Let's create a simple script that uses the 'trap' command:

A simple signal trap

```
#!/bin/sh

trap 'echo Hello World' SIGUSR1

while [ 1 -gt 0 ]
do
   echo Running....
   sleep 5
done
```

This script in and of itself is an endless loop, which prints "Running..." and then sleeps for five seconds. But we've added a 'trap' command (**before** the loop, otherwise the trap would never be executed and it wouldn't affect the loop) that prints "Hello World" whenever the process receives the SIGUSR1 signal. So let's start the process by running the script:

Infinite loop...

Code:

```
$ ./trap_signal.sh
```

Output:

```
Running....
Running....
Running....
Running....
Running....
Running....
...
```

This could get boring after a while....

To spring the trap, we must send the running process a signal. To do that, log into a new shell session and use a process tool (like 'ps') to find the correct process id (PID):

Finding the process ID

Code:

```
$ ps -ef | grep signal
```

Output:

```
bzt 10865 7067 0 15:08 pts/0 00:00:00 /bin/sh ./trap_signal.sh

bzt 10808 10415 0 15:12 pts/1 00:00:00 fgrep signal
```

Our PID is 10865

Now, to send a signal to that process, we use the 'kill' command which is built into the Bourne Shell:

```
kill [-signal] ID [ID] ...
```

*Where *-signal* is the signal to send (optional; default is 15, or SIGTERM)

and *ID* are the PIDs of the processes to send the signal to (at least one of them)

As the name suggests, 'kill' was actually intended to kill processes (this fits with the default signal being SIGTERM and the default signal handler being terminate). But in fact what it does is nothing more than send a signal to a process. So for example, we can send a SIGUSR1 to our process like this:

Let's trip the trap...

Code:

```
kill -SIGUSR1 10865
```

Output:

```
...
Running....
Running....
Running....
Running....
Running....
Hello World
Running....
Running....
...
```

You might notice that there's a short pause before "Hello World!" appears; it won't happen until the running 'sleep' command is done. But after that, there it is. But you might be a little surprised: the signal didn't kill the process. That's because 'trap' *completely* replaces the signal handler with the commands you set up. And an 'echo Hello World' alone won't kill a process... The lesson here is a simple one: if you want your signal trap to terminate your process, make sure you include an 'exit' command.

Between having multiple commands in your command list and potentially trapping lots of signals, you might be worried that a 'trap' statement can become messy. Fortunately, you can also use shell functions as commands in a 'trap'. The following example illustates that and the difference between an exiting event handler and a non-exiting event handler:

A trap with a shell function as a handler

```
#!/bin/sh

exit_with_grace() {
  echo Goodbye World
  exit
}

trap "exit_with_grace" USR1 TERM QUIT
trap "echo Hello World" USR2

while [ 1 -gt 0 ]
do
   echo Running....
   sleep 5
done
```

System signals

Here's the official definition of a signal from the POSIX-1003 2001 edition standard:

```
A mechanism by which a process or thread may be notified of, or affected by, an event occurring in tl
Examples of such events include hardware exceptions and specific actions by processes. The term sign
```

In other words, a signal is some sort of short message that is send from one process (possible a system process) to another. But what does that mean exactly? What does a signal look like? The definition given above is kind of vague...

If you have any feel for what happens in computing when you give a vague definition, you already know the answer to the questions above: every Unix flavor that wa developed came up with its own definition of "signal". They pretty much all settled on a message that consists of an integer (because that's simple), but not exactly the *same* list everywhere. Then there was some standardization and Unix systems organized themselves into the System V and

BSD flavors and at last everybody agreed on the following definition:

```
The system signals are the signals listed in /usr/include/sys/signal.h .
```

God, that's helpful...

Since then a lot has happened, including the definition of the POSIX-1003 standard. This standard, which standardizes most of the Unix interfaces (including the shell in part 1 (1003.1)) finally came up with a standard list of symbolic signal names and default handlers. So usually, nowadays, you can make use of that list and expect your script to work on most systems. Just be aware that it's not *completely* fool-proof...

POSIX-1003 defines the signals listed in the table below. The values given are the *typical* numeric values, but they aren't mandatory and you shouldn't rely on them (but then again, you use symbolic values in order not to use actual values).

POSIX system signals

Signal	Default action	Description	Typical value(s)
SIGABRT	Abort with core dump	Abort process and generate a core dump	6
SIGALRM	Terminate	Alarm clock.	14
SIGBUS	Abort with core dump	Access to an undefined portion of a memory object.	7, 10
SIGCHLD	Ignore	Child process terminated, stopped	20, 17, 18
SIGCONT	Continue process (if stopped)	Continue executing, if stopped.	19,18,25
SIGFPE	Abort with core dump	Erroneous arithmetic operation.	8
SIGHUP	Terminate	Hangup.	1
SIGILL	Abort with core dump	Illegal instruction.	4
SIGINT	Terminate	Terminal interrupt signal.	2
SIGKILL	Terminate	Kill (cannot be caught or ignored).	9
SIGPIPE	Terminate	Write on a pipe with no one to read it (i.e. broken pipe).	13
SIGQUIT	Terminate	Terminal quit signal.	3
SIGSEGV	Abort with core dump	Invalid memory reference.	11
SIGSTOP	Stop process	Stop executing (cannot be caught or ignored).	17,19,23
SIGTERM	Terminate	Termination signal.	15
SIGTSTP	Stop process	Terminal stop signal.	18,20,24
SIGTTIN	Stop process	Background process attempting read.	21,21,26
SIGTTOU	Stop process	Background process attempting write.	22,22,27
SIGUSR1	Terminate	User-defined signal 1.	30,10,16
SIGUSR2	Terminate	User-defined signal 2.	31,12,17
SIGPOLL	Terminate	Pollable event.	-
SIGPROF	Terminate	Profiling timer expired.	27,27,29
SIGSYS	Abort with core dump	Bad system call.	12
SIGTRAP	Abort with core dump	Trace/breakpoint trap	5
SIGURG	Ignore	High bandwidth data is available at a socket.	16,23,21
SIGVTALRM	Terminate	Virtual timer expired.	26,28
SIGXCPU	Abort with core dump	CPU time limit exceeded.	24,30
SIGXFSZ	Abort with core dump	File size limit exceeded.	25,31

Earlier on we talked about job control and suspending and resuming jobs. Job suspension and resuming is actually completely based on sending signals to processes, so you can in fact control job stopping and starting completely using 'kill' and the signal list. To suspend a process, send it the SIGSTOP signal. To resume, send it the SIGCONT signal.

Err... ERR?

If you go online and read about 'trap', you might come across another kind of "signal" which is called **ERR**. It's used with 'trap' the same way regular signals are, but it isn't really a signal at all. It's used to trap command errors (i.e. non-zero exit statuses), like this:

Error trapping

Code:

```
$ trap 'echo HELLO WORLD' ERR
$ expr 1 / 0
```

Output:

```
expr: division by zero

HELLO WORLD
```

The non-zero exit status was trapped as though it was a signal.

So why didn we cover this "signal" earlier, when we were discussing 'trap'? Well, we saved it until the discussion on system and non-system signals for a reason: ERR isn't standard at all. It was added by the Korn Shell to make life easier, but not adopted by the POSIX standard and it certainly isn't part of the original Bourne Shell. So if you use it, remember that your script may not be portable anymore.

Cookbook

Post a new Cookbook entry (http://en.wikibooks.org /w/index.php?title=Bourne_Shell_Scripting

/Print_Version&action=edit&section=new)

If you use the title box, then you do not need to put a title in the body.

Branch on extensions

When writing a bash script which should do different things based on the the extension of a file, the following pattern is helpful.

```
#filepath should be set to the name(with optional path) of the file in question
ext=${filepath##*.}
if [[ "$ext" == txt ]] ; then
  #do something with text files
fi
```

(Source: slike.com Bash FAQ (http://www.splike.com/howtos/bash_faq.html#Get+a+file%27s+basename%2C+dirname%2C+extension%2C+etc%3F)).

Rename several files

This recipe shows how to rename several files following a pattern.

In this example, the user has huge collection of screenshots. This user wants to rename the files using a Bourne-compatible shell. Here is an "ls" at the shell prompt to show you the filenames. The goal is to rename images like "snapshot1.png" to "nethack-kernigh-22oct2005-01.png".

```
$ ls
snapshot1.png   snapshot25.png  snapshot40.png  snapshot56.png  snapshot71.png
snapshot10.png  snapshot26.png  snapshot41.png  snapshot57.png  snapshot72.png
snapshot11.png  snapshot27.png  snapshot42.png  snapshot58.png  snapshot73.png
snapshot12.png  snapshot28.png  snapshot43.png  snapshot59.png  snapshot74.png
snapshot13.png  snapshot29.png  snapshot44.png  snapshot6.png   snapshot75.png
snapshot14.png  snapshot3.png   snapshot45.png  snapshot60.png  snapshot76.png
snapshot15.png  snapshot30.png  snapshot46.png  snapshot61.png  snapshot77.png
snapshot16.png  snapshot31.png  snapshot47.png  snapshot62.png  snapshot78.png
snapshot17.png  snapshot32.png  snapshot48.png  snapshot63.png  snapshot79.png
snapshot18.png  snapshot33.png  snapshot49.png  snapshot64.png  snapshot8.png
snapshot19.png  snapshot34.png  snapshot5.png   snapshot65.png  snapshot80.png
snapshot2.png   snapshot35.png  snapshot50.png  snapshot66.png  snapshot81.png
snapshot20.png  snapshot36.png  snapshot51.png  snapshot67.png  snapshot82.png
snapshot21.png  snapshot37.png  snapshot52.png  snapshot68.png  snapshot83.png
snapshot22.png  snapshot38.png  snapshot53.png  snapshot69.png  snapshot9.png
snapshot23.png  snapshot39.png  snapshot54.png  snapshot7.png
snapshot24.png  snapshot4.png   snapshot55.png  snapshot70.png
```

First, to add a "0" (zero) before snapshots 1 through 9, write a for loop (in effect, a short shell script).

- Use `?` which is a filename pattern for a single character. Using it, I can match snapshots 1 through 9 but miss 10 through 83 by saying `snapshot?.png`.
- Use `${`*parameter#pattern*`}` to substitute the value of *parameter* with the *pattern* removed from the beginning. This is to get rid of "snapshot" so I can put in "snapshot0".
- Before actually running the loop, insert an "echo" to test that the commands will be correct.

```
$ for i in snapshot?.png; do echo mv "$i" "snapshot0${i#snapshot}"; done
mv snapshot1.png snapshot01.png
mv snapshot2.png snapshot02.png
mv snapshot3.png snapshot03.png
mv snapshot4.png snapshot04.png
mv snapshot5.png snapshot05.png
mv snapshot6.png snapshot06.png
mv snapshot7.png snapshot07.png
mv snapshot8.png snapshot08.png
mv snapshot9.png snapshot09.png
```

That seems good, so run it by removing the "echo".

```
$ for i in snapshot?.png; do mv "$i" "snapshot0${i#snapshot}"; done
```

An ls confirms that this was effective.

Now change prefix "snapshot" to "nethack-kernigh-22oct2005-". Run a loop similar to the previous one:

```
$ for i in snapshot*.png; do
> mv "$i" "nethack-kernigh-22oct2005-${i#snapshot}"
> done
```

This saves the user from typing 83 "mv" commands.

Long command line options

The builtin `getopts` does not support long options so the external `getopt` is required. (On some systems, `getopt` *also* does not support long options, so the next example will not work.)

```
eval set -- $(getopt -l install-opts: "" "$@")
while true; do
    case "$1" in
        --install-opts)
            INSTALL_OPTS=$2
            shift 2
            ;;
        --)
            shift
            break
            ;;
    esac
done

echo $INSTALL_OPTS
```

The call to `getopt` quotes and reorders the command line arguments found in `$@`. `set` then makes replaces `$@` with the output from `getopt`

Another example of getopt use can also be found in the Advanced Bash Script Guide (http://www.tldp.org/LDP/abs/html/extmisc.html#EX33A)

Process certain files through xargs

In this recipe, we want to process a large list of files, but we must run one command for each file. In this example, we want to convert the sampling rates of some sound files to 44100 hertz. The command is `sox file.ogg -r 44100 conv/file.ogg`, which converts `file.ogg` to a new file `conv/file.ogg`. We also want to skip files that are already 44100 hertz.

First, we need the sampling rates of our files. One way is to use the `file` command:

```
$ file *.ogg
audio_on.ogg:            Ogg data, Vorbis audio, mono, 44100 Hz, ~80000 bps
beep_1.ogg:              Ogg data, Vorbis audio, stereo, 44100 Hz, ~193603 bps
cannon_1.ogg:            Ogg data, Vorbis audio, mono, 48000 Hz, ~96000 bps
...
```

(The files in this example are from Secret Maryo Chronicles (http://www.secretmaryo.org/) .) We can use `grep -v` to filter out all lines that contain '44100 Hz':

```
$ file *.ogg | grep -v '44100 Hz'
cannon_1.ogg:            Ogg data, Vorbis audio, mono, 48000 Hz, ~96000 bps
...
jump_small.ogg:          Ogg data, Vorbis audio, mono, 8000 Hz, ~22400 bps
live_up.ogg:             Ogg data, Vorbis audio, mono, 22050 Hz, ~40222 bps
...
```

We finished with "grep" and "file", so now we want to remove the other info and leave only the filenames to pass to "sox". We use the text utility `cut`. The option `-d:` divides each line into fields at the colon; `-f1` selects the first field.

```
$ file *.ogg | grep -v '44100 Hz' | cut -d: -f1
cannon_1.ogg
...
jump_small.ogg
live_up.ogg
...
```

We can use another pipe to supply the filenames on the standard input, but "sox" expects them as arguments. We use `xargs`, which will run a command repeatedly using arguments from the standard input. The `-n1` option specifies one argument per command. For example, we can run `echo sox` repeatedly:

```
$ file *.ogg | grep -v '44100 Hz' | cut -d: -f1 | xargs -n1 echo sox
sox cannon_1.ogg
...
sox itembox_set.ogg
sox jump_small.ogg
...
```

However, these commands are wrong. The full command for cannon_1.ogg, for example, is `sox cannon_1.ogg -r 44100 conv/cannon_1.ogg`. "xargs" will insert incoming data into placeholders indicated by "{}". We use this strategy in our pipeline. If we have doubt, then first we can build a test pipeline with "echo":

```
$ file *.ogg | grep -v '44100 Hz' | cut -d: -f1 | \
> xargs -i 'echo sox {} -r 44100 conv/{}'
sox cannon_1.ogg -r 44100 conv/cannon_1.ogg
...
sox itembox_set.ogg -r 44100 conv/itembox_set.ogg
sox jump_small.ogg -r 44100 conv/jump_small.ogg
...
```

It worked, so let us remove the "echo" and run the "sox" commands:

```
$ mkdir conv
$ file *.ogg | grep -v '44100 Hz' | cut -d: -f1 | \
> xargs -i 'sox {} -r 44100 conv/{}'
```

After a wait, the converted files appear in the `conv` subdirectory. The above three lines alone did the entire conversion.

Simple playlist frontend for GStreamer

If you have GStreamer, the command `gst-launch filesrc location=`*`filename`*` ! decodebin ! audioconvert ! esdsink` will play a sound or music file of any format for which you have a GStreamer plugin. This script will play through a list of files, optionally looping through them. (Replace "esdsink" with your favorite sink.)

```
#!/bin/sh
loop=false
if test x"$1" == x-l; then
  loop=true
  shift
fi

while true; do
  for i in "$@"; do
    if test -f "$i"; then
      echo "${0##*/}: playing $i" > /dev/stderr
      gst-launch filesrc location="$i" ! decodebin ! audioconvert ! esdsink
    else
      echo "${0##*/}: not a file: $i" > /dev/stderr
    fi
  done
  if $loop; then true; else break; fi
done
```

This script demonstrates some common Bourne shell tactics:

- "loop" is a boolean variable. It works because its values "true" and "false" are both Unix commands (and sometimes shell builtins), thus you can use them as conditions in `if` and `while` statements.
- The shell builtin "shift" removes $1 from the argument list, thus shifting $2 to $1, $3 to $2, and so forward. This script uses it to process an "-l" option.
- The substitution `${0##*/}` gives everything in $0 after the last slash, thus "playlist", not "/home/musicfan/bin/playlist".

Quick Reference

This final section provides a fast lookup reference for the materials in this document. It is a collection of thumbnail examples and rules that will be cryptic if you haven't read through the text.

Useful commands

Command	Effect
cat	Lists a file or files sequentially.
cd	Change directories.
chmod ugo+rwx	Set read, write and execute permissions for user, group and others.
chmod a-rwx	Remove read, write and execute permissions from all.
chmod 755	Set user write and universal read-execute permissions

chmod 644	set user write and universal read permissions.
cp	Copy files.
expr 2 + 2	Add 2 + 2.
fgrep	Search for string match.
grep	Search for string pattern matches.
grep -v	Search for no match.
grep -n	List line numbers of matches.
grep -i	Ignore case.
grep -l	Only list file names for a match.
head -n5 source.txt	List first 5 lines.
less	View a text file one screen at a time; can scroll both ways.
ll	Give a listing of files with file details.
ls	Give a simple listing of files.
mkdir	Make a directory.
more	Displays a file a screenfull at a time.
mv	Move or rename files.
paste f1 f2	Paste files by columns.
pg	Variant on "more".
pwd	Print working directory.
rm	Remove files.
rm -r	Remove entire directory subtree.
rmdir	Remove an empty directory.
sed 's/txt/TXT/g'	Scan and replace text.
sed 's/txt/d'	Scan and delete text.
sed '/txt/q'	Scan and then quit.
sort	Sort input.
sort +1	Skip first field in sorting.
sort -n	Sort numbers.

sort -r	Sort in reverse order.
sort -u	Eliminate redundant lines in output.
tail -5 source.txt	List last 5 lines.
tail +5 source.txt	List all lines after line 5.
tr '[A-Z]' '[a-z]'	Translate to lowercase.
tr '[a-z]' '[A-Z]'	Translate to uppercase.
tr -d '_'	Delete underscores.
uniq	Find unique lines.
wc	Word count (characters, words, lines).
wc -w	Word count only.
wc -l	Line count.

Elementary shell capabilities

Command	Effect
shvar="Test 1"	Initialize a shell variable.
echo $shvar	Display a shell variable.
export shvar	Allow subshells to use shell variable.
mv $f ${f}2	Append "2" to file name in shell variable.
$1, $2, $3, ...	Command-line arguments.
$0	Shell-program name.
$#	Number of arguments.
$*	Complete argument list (all in one string).
$@	Complete argument list (string for every argument).
$?	Exit status of the last command executed.
shift 2	Shift argument variables by 2.
read v	Read input into variable "v".
. mycmds	Execute commands in file.

IF statement

The if statement executes the command between `if` and `then`. If the command returns not 0 then the commands between `then` and `else` are executed - otherwise the command between `else` and `fi`.

```
if test "${1}" = "red" ; then
  echo "Illegal code."
elif test "${1}" = "blue" ; then
  echo "Illegal code."
else
  echo "Access granted."
fi
```

```
if [ "$1" = "red" ]
then
  echo "Illegal code."
elif [ "$1" = "blue" ]
then
  echo "Illegal code."
else
  echo "Access granted."
fi
```

Test Syntax Variations

Most test commands can be written using more then one syntax. Mastering and consistently using one form may be a programming best-practice, and may be a more efficient use of overall time.

String Tests

String Tests are performed by the `test` command. See `help test` for more details. To make scripts look more like other programming languages the synonym `[ ... ]` was defined which does exactly the same as `test`.

Command	Effect
test "$shvar" = "red" ["$shvar" = "red"]	String comparison, true if match.
test "$shvar" != "red" ["$shvar" != "red"]	String comparison, true if no match.

test -z "${shvar}" test "$shvar" = "" ["$shvar" = ""]	True if null variable.
test -n "${shvar}" test "$shvar" != "" [-n "$shvar"] ["$shvar" != ""]	True if not null variable.

Arithmetic tests

simple arithmetics can be performed with the `test` for more complex arithmetics the `let` command exists. See `help let` for more details. Note that for `let` command variables don't need to be prefixed with '$' and the statement need to be one argument, use `'...'` when there are spaces inside the argument. Like with `test` a synonym - `(( ... ))` - was defined to make shell scripts look more like ordinary programs.

Command	Effect
test "$nval" -eq 0 let 'nval == 0' ["$nval" -eq 0] ((nval == 0))	Integer test; true if equal to 0.
test "$nval" -ge 0 let 'nval >= 0' ["$nval" -ge 0] ((nval >= 0))	Integer test; true if greater than or equal to 0.
test "$nval" -gt 0 let 'nval > 0' ["$nval" -gt 0] ((nval > 0))	Integer test; true if greater than 0.
test "$nval" -le 0 let 'nval <= 0'	Integer test; true if less than or equal to 0.

["$nval" -le 0] ((nval <= 0))	
test "$nval" -lt 0 let 'nval < 0' ["$nval" -lt 0] ((nval < 0))	Integer test; true if less than to 0.
test "$nval" -ne 0 let 'nval != 0' ["$nval" -ne 0] ((nval != 0))	Integer test; true if not equal to 0.
let 'y + y > 100' ((y + y >= 100))	Integer test; true when $x + y \geq 0$

File tests

Command	Effect
test -d tmp [-d tmp]	True if "tmp" is a directory.
test -f tmp [-f tmp]	True if "tmp" is an ordinary file.
test -r tmp [-r tmp]	True if "tmp" can be read.
test -s tmp [-s tmp]	True if "tmp" is nonzero length.
test -w tmp	True if "tmp" can be written.

[-w tmp]	
test -x tmp [-x tmp]	True if "tmp" is executable.

Boolean tests

Boolean arithmetic is performed by a set of operators. It is important to note then the operators execute programs and compare the result codes. Because boolean operators are often combined with `test` command a unifications was created in the form of `[[ ... ]]`.

Command	Effect
test -d /tmp && test -r /tmp [[-d /tmp && -r /tmp]]	True if "/tmp" is a directory and can be read.
test -r /tmp || test -w /tmp [[-r /tmp || -w /tmp]]	True if "tmp" can be be read or written.
test ! -x /tmp [[! -x /tmp]]	True if the file is not executable

CASE statement

```
case "$1"
in
  "red")      echo "Illegal code."
              exit;;
  "blue")     echo "Illegal code."
              exit;;
  "x"|"y")    echo "Illegal code."
              exit;;
  *)          echo "Access granted.";;
esac
```

Loop statements

```
for nvar in 1 2 3 4 5
do
  echo $nvar
done
```

```
for file            # Cycle through command-line arguments.
do
  echo $file
done
```

```
while [ "$n" != "Joe" ]     # Or:   until [ "$n" = "Joe" ]
do
  echo "What's your name?"
  read n
  echo $n
done
```

There are "break" and "continue" commands that allow you to exit or skip to the end of loops as the need arises.

Instead of [] we can use test. [] requires space after and before the brackets and there should be spaces between arguments.

Credit

This content was originally from http://www.vectorsite.net/tsshell.html and was originally in the public domain.

Appendix A: Command Reference

The Bourne Shell offers a large number of built-in commands that you can use in your shell scripts. The following table gives an overview:

Bourne Shell command reference

Command	Description
:	A null command that returns a 0 (true) exit value.
. *file*	Execute. The commands in the specified file are read and executed by the shell. Commonly referred to as *sourcing* a file.
#	Ignore all the text until the end of the line. Used to create comments in shell scripts.
#!*shell*	Interpreter hint. Indicates to the OS which interpreter to use to execute a script.

bg [job] ...	Run the specified jobs (or the current job if no arguments are given) in the background.
break [n]	Break out of a loop. If a number argument is specified, break out of n levels of loops.
case	See Bourne Shell Scripting/Control flow
cd [directory]	Switch to the specified directory (default $HOME).
continue [n]	Skip the remaining commands in a loop and continue the loop at the next interation. If an integer argument is specified, skip n loops.
echo string	Write string to the standard output.
eval string ...	Concatenate all the arguments with spaces. Then re-parse and execute the command.
exec [command arg ...]	Execute command in the current process.
exit [exitstatus]	Terminate the shell process. If exitstatus is given it is used as the exit status of the shell; otherwise the exit status of the last completed command is used.
export name ...	Mark the named variables or functions for export to child process environments.
fg [job]	Move the specified job (or the current job if not specified) to the foreground.
for	See Bourne Shell Scripting/Control flow.
hash -rv command ...	The shell maintains a hash table which remembers the locations of commands. With no arguments whatsoever, the hash command prints out the contents of this table. Entries which have not been looked at since the last cd command are marked with an asterisk; it is possible for these entries to be invalid. With arguments, the hash command removes the specified commands from the hash table (unless they are functions) and then locates them. The -r option causes the hash command to delete all the entries in the hash table except for functions.
if	See Bourne Shell Scripting/Control flow.
jobs	This command lists out all the background processes which are children of the current shell process.
-*signal*] PID ...	Send signal to the jobs listed by ID. If no signal is specified, send SIGTERM. If the -l option is used, lists all the signal names defined on the system.

newgrp [group]	Temporarily move your user to a new group. If no group is listed, move back to your user's default group.
pwd	Print the working directory.
read variable [...]	Read a line from the input and assign each individual word to a listed variable (in order). Any leftover words are assigned to the last variable.
readonly name ...	Make the listed variables read-only.
return [n]	Return from a shell function. If an integer argument is specified it will be the exit status of the function.
set [{ -options | +options | -- }] arg ...	The set command performs three different functions. With no arguments, it lists the values of all shell variables. If options are given, it sets the specified option flags or clears them. The third use of the set command is to set the values of the shell's positional parameters to the specified args. To change the positional parameters without changing any options, use “--” as the first argument to set. If no args are present, the set command will clear all the positional parameters (equivalent to executing “shift $#”.)
shift [n]	Shift the positional parameters n times.
test	See Bourne Shell Scripting/Control flow.
trap [action] signal ...	Cause the shell to parse and execute action when any of the specified signals are received.
type [name ...]	Show whether a command is a UNIX command, a shell built-in command or a shell function.
ulimit	Report on or set resource limits.
umask [mask]	Set the value of umask (the mask for the default file permissions *not* assigned to newly created files). If the argument is omitted, the umask value is printed.
unset name ...	Drop the definition of the given names in the shell.
wait [job]	Wait for the specified job to complete and return the exit status of the last process in the job. If the argument is omitted, wait for all jobs to complete and the return an exit status of zero.
while	See Bourne Shell Scripting/Control flow.

Appendix B: Environment reference

In the section on the environment we discussed the concept of environment variables. We also mentioned that there are usually a large number of environment variables that are created centrally in **/etc/profile**. There are a number of these that have a predefined meaning in the Bourne Shell. They are not set automatically, mind, but they have meaning when they are set.

On most systems there are far more predefined variables than we list here. And some of these will mean something to your shell (most shells have more options than the Bourne Shell). Check your shell's documentation for a listing. The ones below are meaningful to the Bourne Shell and are usually also recognized by other shells.

Bourne Shell environment variables

Variable	**Meanings**
HOME	The user's home directory. Set automatically at login from the user's login directory in the password file
PATH	The default search path for executables.
CDPATH	The search path used with the cd builtin, to allow for shortcuts.
LANG	The directory for internationalization files, used by localizable programs.
MAIL	The name of a mail file, that will be checked for the arrival of new mail.
MAILCHECK	The frequency in seconds that the shell checks for the arrival of mail.
MAILPATH	A colon “:” separated list of file names, for the shell to check for incoming mail.
PS1	The control string for your prompt, which defaults to “$ ”, unless you are the superuser, in which case it defaults to “# ”.
PS2	The control string for your secondary prompt, which defaults to “> ”. The secondary prompt is what you see when you break a command over more than one line.
PS4	The character string you see before the output of an execution trace (set -x); defaults to “+ ”.
IFS	Input Field Separators. Basically the characters the shell considers to be whitespace. Normally set to 〈space〉, 〈tab〉, and 〈newline〉.
TERM	The terminal type, for use by the shell.