Shell Basics

As noted on page 116 in your book, the fundamental process for an interactive shell is

- The shell first displays some sort of "prompt".
- You type some command, which can either be a "built-in" (i.e., the shell just does what you ask rather than starting a separate child process), or can name a program to be executed as a separate process.

- Whatever you type, when you hit return, the shell first takes a good long look at your input: it evaluates and substitutes for any metacharacters, it expands any variables, and then it tries to match the command against its internal command set. If it doesn't find anything there, it finally looks around the directories listed in the $PATH variable to see if it can find a program of that name.
- Finally, once the program is complete (or you put it in the "background"), you get a new prompt.

Environment variables and their effects

You can get a list of the current shell's environmental variables in various ways: `env`, `printenv`; (in Bash, you can also use `set`, but that also gives additional shell variables that aren't actually in the process's environment.)

Important environment variables:

- PATH
- HOME
- USER
- SHELL

Local variables

In addition to environmental variables which are passed on to child processes, you can create local variables.

```
% export x=12    # create an environmental variable
% y=15           # create a local variable
% echo x = $x and y = $y
x = 12 and y = 15
% bash           # now create a new child process
% echo x = $x and y = $y
x = 12 and y =
```

Prompts

Bourne shell, Korn shell, Bash, and others in that family have two primary prompt variables, PS1 and PS2

```
export PS1="# "
export PS2="... "
```

Redirection

I have already mentioned input redirection in the context of creating files. Overall, there are three important redirection operators: <, >, and >>

The < operator lets you redirect standard in; for instance, you can do

```
sort < /etc/passwd
```

and this will have sort takes its standard in from the file `/etc/passwd`.

As we have seen already, the > operator lets you redirect standard out; for instance, you can take the output of `cal` and save it:

```
cal 1752 > /tmp/unusual-cal.txt
```

Finally, you can use the >> operator to append data to a file:

```
cal 1753 >> /tmp/unusual-cal.txt
```

Named file descriptors

You can also specifically name which file descriptor to use with the form "n>" and "n<".

This is particularly useful when you want to split, say, stdout and stderr data out to two different places.

```
(ls -R | wc -l) 2>/dev/null
```

Merging file descriptor data

You can also merge file descriptor data with the special forms "x>&y" and "x<&y". The first lets you merge the output of two file descriptors. For example:

```
ls -lR / 1>&2
```

This sends all of file descriptor 1 (stdout) also to file descriptor 2 (stderr).

"Here" documents

Another very useful type of redirection, particularly in shell scripts, is the "here" document.

The syntax looks like

```
<<'EOF'
data... data...
data... data...
EOF
```

For example:

```
$ cat <<'EOF'
> this is stuff that
> we want to echo
> until the
> EOF
this is stuff that
we want to echo
until the
$
```

In addition to simple file redirection, we have the concept of a "pipe", which is a creation of the kernel. It's actually just a simple "buffer".

The syntax for creating a pipe between two processes is the vertical bar ("|").

```
sort < /etc/passwd | cut -d: -f 1
```

The `tee`: there's a nice program call `tee` that will let you intercept "mid-pipe" data.

```
$ sort < /etc/passwd | tee /tmp/sorted-passwd | cut -d: -f 1
$ cat /tmp/sorted-passwd
```

More uses for data from processes

Unix has had long had the very useful concept of being able to expand output from a process into the command line of a shell process.

```
echo The date and time is `date`
```

This can be particularly valuable when you save the output to a variable:

```
% x=`date --iso-8601`
% mkdir new-$x/ old-$x/ cur-$x/
% ls -d *$x
cur-2013-01-29   new-2013-01-29   old-2013-01-29
%
```

Filters

Once we have the concept of a bytestream flowing through pipes, a natural metaphor for programs that modify the bytestream is to call them a "filter".

There are a very large number of standard filters; some of the most useful are here.

Here's a filter example derived from page 107 of Kernighan and Pike's *The Unix Programming Environment* that shows the 8 most frequently used words in *Pilgrim's Progress* by John Bunyan (book contents courtesy of Project Gutenberg):

```
% tr -sc A-Za-z '\012' < pg39452.txt   | sort | uniq -c | sort -n | tail -8
   1454 in
   1567 a
   1910 I
   2289 that
   2570 of
   3078 to
   3479 and
   4397 the
```

Here's a "one-liner" I created to list all of the system calls on a given system:

```
% ls /usr/share/man/man2 | sed -e s/.2.gz//g | \
> xargs man -s 2 -k  | sort | \
> grep -v 'unimplemented system calls'
```

Foreground/background

One of the great strengths of Unix has been its very clean "job control". You can easily send a process into the background with a simple ampersand:

```
% ( ls -R / | wc -l ) 2>/dev/null &
% jobs
[1]+  Running                 ( ls -R / | wc -l ) 2> /dev/null &
```

You can start many jobs:

```
% cat &
[1] 7200
% cat &
[2] 7201
% cat &
[3] 7203
% cat &
[4] 7209
% fg %1      # now bring job #1 back
cat
some input
some input
[CTRL-D]
% jobs       # okay, we see that job #1 has finished
[2]   Stopped                 cat
[3]-  Stopped                 cat
[4]+  Stopped                 cat
% fg %3      # now bring job #3
cat
more input
more input
[CTRL-D]
% jobs       # and now we see that it's finished also
[2]-  Stopped                 cat
[4]+  Stopped                 cat
%           # and so forth...
```

Killing a stubborn job (think LaTeX!)

If you get stuck in a LaTeX session (or emacs, for that matter), you can often use CTRL-Z to get out. Once you do, you can use `kill` to get rid of it:

```
pdflatex file1.tex
This is pdfTeX, Version 3.1415926-1.40.10 (TeX Live 2009/Debian)
entering extended mode
! I can't find file `file1.tex'.
<*> file1.tex

(Press Enter to retry, or you will never exit!)
Please type another input file name:

[1]+  Stopped                 pdflatex file1.tex
[CTRL-Z]
% kill -9 %1
```