Chapter 8

The Shell (I)

When you type commands, they are interpreted by a command *shell.* The shell is a program that interprets user commands before passing them along to be executed. Command shells are quite sophisticated and provide a number of useful features. Although there is a lot to learn about shells, we will explore only those features that facilitate use of Humdrum. This chapter is the first of four chapters scattered throughout this book where we will pause and examine some of the more pertinent and valuable features of the shell.

In UNIX environments, many different shells have been developed over the years. The original UNIX shell was the *C-shell* — a shell whose syntax is similar to the C programming language. A later shell was developed by Stephen Bourne and is known as the *Bourne shell.* Subsequent improvements by David Korn resulted in the *Korn shell.* The Bourne shell was improved in light of many features introduced in the Korn shell, and resulted in the *Bourne Again Shell* — known as *Bash.* The Korn and Bash shells are the most popular and powerful of the current generation of shells. Although they were originally developed for the UNIX operating system, these shells are also available for DOS, Macintosh, Windows, Windows 98 and many other operating systems.

Shells themselves are advanced programming languages that provide complex control structures. When you type a command, you are already writing a program — although most of your programs are just one line in length.

Shell Special Characters

The shell interprets a number of characters in a special manner. When you type a command, you should know that most shells treat the following characters as having a special meaning: the octothorpe (#), the dollar-sign ($), the semicolon (;), the ampersand (&), the verticule (|), the asterisk (*), the apostrophe ('), the greve (‘), the greater-than sign (>), the less-than sign (<), the question-mark (?), the double-quote ("), and the backslash (\). We'll consider the function of each of these characters one at a time.

File Redirection (>)

Some of the special shell characters have already been discussed. The greater-than-sign (>) is a *file redirection operator.* It must be followed by a user-specified filename; any output from the preceding command is placed in the specified file. For example, the following command sorts the file `inputfile` and places the sorted result in the file named `outputfile`:

```
sort inputfile > outputfile
```

If the file `outputfile` already existed, its contents will be destroyed and over-written with the new output. Be careful not to assign the output to the same file as the input, since this will destroy the original input file.

Sometimes it is useful to *add* the results of an operation to some already existing file. The double greater-than-sign (>>) causes the new output to be appended to any data already in the named file. For example, the following command sorts the file `inputfile` and adds the sorted lines to the end of the file named `outputfile`. If the `outputfile` does not already exist, the command will create it.

```
sort inputfile >> outputfile
```

Pipe (|)

The vertical bar (|) is interpreted by the shell as a 'pipe.' Pipes are used to join the output of one command to the input of a subsequent command. For example, in the following construction, the output of `command1` is routed as the input to `command2`:

```
command1 | command2
```

There is no practical limit to the length of a pipeline. Several pipes can be used to connect successive outputs to ensuing commands:

```
command1 | command2 | command3 | command4
```

Shell Wildcard (*)

The asterisk is interpreted by the shell as a "filename wildcard." When it appears by itself, the asterisk is 'expanded' by the shell to a list of all files in the current directory (in alphabetical order). For example, if the current directory contained just three files: `alice`, `barry` and `chris` — then the following command would be applied to all three files in consecutive order:

```
command * > people
```

The file expansion occurs at the moment when the command is invoked. So although the file `people` is added to the current directory, it is not included as its own input. However, if the above command was executed a second time, then the file expansion would include `people` — even as the file itself is over-written to receive the output. Including the output file as input is never a good idea.

Comment (#)

The octothorpe character (#) indicates a shell *comment.* Any characters following the # (up to the end of the line) are simply ignored by the shell. The following is not a command:

```
#grep OTL: filename
```

The comment can begin anywhere in the line. Here the comment begins after the filename:

```
grep OTL: filename   # (Search for Humdrum titles.)
```

Escape Character (\)

Sometimes we would like to have a special character treated literally. For example, suppose we wanted to search for records containing sharps in a `**kern` file. The following command will not work because the shell will insist on interpreting the octothorpe as beginning a comment:

```
grep # filename
```

There are several ways to "turn off" the special meaning of a character. The simplest way is to precede the character by a backslash (\) as in the following command:

```
grep \# filename
```

The backslash character itself can be treated literally by preceding it with another backslash. For example, the following command searches for down-stems in a `**kern` file:

```
grep \\ filename
```

Escape Quotations ('...')

Another way of escaping the special meaning of shell characters is to place the material in single quotes. For example, we can escape the meaning of the octothorpe (#) by preceding and following it by single quotes:

```
grep '#' filename
```

Single quotes are especially useful for binding spaces. For example, the following command searches for the phrase "Lennon and McCartney" in a file named `beatles`:

```
grep 'Lennon and McCartney' beatles
```

If the single quotes are omitted, the command means something completely different. The following command searches for the string "Lennon" in three files named `and,` `McCartney` and `bea-`

`tles:`

```
grep Lennon and McCartney beatles
```

A common mistake is to fail to match quotation marks in a command. The shell will assume that the command is incomplete until all quotation marks are matched (both single quotes and double quotes). In the following example, we have failed to match the quotation mark. When we press the return key, the shell responds with a change of prompt indicating that it is waiting for us to complete the command.

```
grep '# inputfile > outputfile
>
```

Command Delimiter (;)

The semicolon (;) indicates the end of a command. Its presence allows more than one command to be typed on a single line. For example, the following line:

```
command1 ; command2
```

is logically identical to:

```
command1
command2
```

When both commands appear on the same line, they are still executed sequentially, so the second command doesn't begin until the first is completed. Although the ability to place two or more commands on a single line may seem redundant, there are a number of circumstances where this feature proves useful.

Background Command (&)

After typing a command, the command begins executing as soon as you type the carriage return or "enter" key. When the command has finished executing, the shell will display a new command prompt. Sometimes a command can take a long time to execute so it will be awhile before the prompt is displayed again. Unfortunately, you must wait for the prompt before you can type a new command. On multitasking systems it is possible for the computer to execute more than one command concurrently. The ampersand (&) can be used to execute a command as a *background process*. When a command is ended by an ampersand, the shell creates an independent process to handle the command, and the shell immediately returns with a prompt for a new command from the user. UNIX systems provide sophisticated mechanisms for controlling concurrent processing of commands. For further information concerning these features, refer to a UNIX reference book.

Shell Command Syntax

Shell commands follow a special syntax. There are six possible components to a common command:

1. the command name,
2. one or more options,
3. one or more option parameters,
4. a command argument,
5. one or more input file names,
6. output redirection.

Each of these components is separated by 'blank space' (tabs or spaces). A command begins with the command name — such as **uniq**, **sort**, or **pitch**. A command argument is a special requirement of only some commands. A good example of a command argument is the search pattern given to the **grep** command. In the following command, **grep** is the command name, "Lennon" is the command argument and `beatles` is the input file name:

```
grep Lennon beatles
```

For most commands, it is possible to process more than one input file. These files are simply listed at the end of the command. For example, the following **grep** command searches for the string "McCartney" in the file `beatles` and in the file `wings`:

```
grep McCartney beatles wings
```

Most commands provide *options* that modify the behavior of the command in some way. Command options are designated by a leading dash character. The specific option is usually indicated by a single alphabetic letter, such as the **-b** option (spoken: "dash-B" option). In the **uniq** command, the **-c** option causes a count to be prepended to each output line. In the following command, **uniq** is the command name, **-c** is the option, and `ghana32` is the name of the input file:

```
uniq -c ghana32
```

In many cases, the option is followed by a *parameter* that specifies further information pertaining to the invoked option. In the following command, **recode** is the command name, **-f** is the option, **reassign** is the parameter used by the **-f** option, and **gagaku** is the name of the input file:

```
recode -f reassign gagaku
```

Options and their accompanying parameters must be separated by blank space (i.e. one or more spaces and/or tabs). If more than one option is invoked, and none of the invoked options require a parameter, then the option-letters may be combined. For example, the **-a** and **-b** options might be invoked as **-ab** (or as **-ba)** — provided neither option requires a parameter.

Whenever an option requires a parameter, the option must be specified alone and followed immediately by the appropriate parameter. For example, in the following command, the command name is **trans**, the **-d** option is followed by the numerical parameter **3**; the parameter for the **-c** option is the number **4** and the input file is named **gambia21**.

```
trans -d 3 -c 4 gambia21
```

Since numerical parameters can sometimes be negative, it can be difficult to discern whether a negative number is a parameter or another option. In the following example, the **−3** is a parameter to the **-d** option rather than an option by itself.

```
trans -d -3 -c 2 gambia21
```

Output Redirection

Most commands support several input and output modes. Input to a command may come from three sources. In many cases the input will come from one or more existing files. Apart from existing files, input may also come from text typed manually at the terminal, or from the output of preceding commands. When input text is entered manually it must be terminated with an end-of-file character (control-D) on a separate line. (On Microsoft operating systems the end-of-file character is control-Z.) When input is received from preceding commands, the output is sent via a UNIX pipe ('|') as discussed above.

The different ways of providing input to a command are illustrated in the following examples. In the first example, the input (if any) is taken from the terminal (keyboard). In the second example, the input is *explicitly* taken from a file named `input`. In the third example, the input is *implicitly* taken from a file named `input`. In the fourth example, the input to **command2** comes from the output of **command1**.

```
command
command < input
command input
command1 | command2
```

Outputs produced by commands may similarly be directed to a variety of locations. The default output from most commands is sent to the terminal screen. Alternatively, the output can be sent to another process (i.e. another command) using a pipe (|). Output can also be stored in a file using file redirection operator ('>') or *added* to the end of a (potentially) existing file using the file-append operator ('>>'). In the first example below, the output is sent to the screen. In the second example, the output is sent to the file `outfile`; if the file `outfile` already exists, its contents will be overwritten. In the third example, the output is appended to the end of the file `outfile`; if the file `outfile` does not already exist, it will be created. In the fourth example, the output is sent as input to the command **command2**.

```
command
command > outfile
command >> outfile
command1 | command2
```

When two or more commands have their inputs and outputs linked together using the pipe operator (|), the entire command line is known as a *pipeline.* Pipelines occur frequently in Humdrum applications.

Tee

A special shell command known as **tee** can be used to clone a copy of some output, so that two identical output streams are generated. In the first example below, the output is piped to **tee** which writes one copy of the output to the file `outfile` and the second copy appears on the screen. In the second example, the output from **command1** is split: one copy is piped to **command2** for further processing, while an identical copy is stored in the file `outfile1`; if the file `outfile1` already exists, its contents will be overwritten. In the third example, the append option (**-a**) for **tee** has been invoked — meaning that the output from `command` will be added to the end of any existing data in the file `outfile`. If the file `outfile` does not already exist, it will be created.

```
command | tee outfile
command1 | tee outfile1 | command2 > outfile2
command | tee -a outfile
```

The **tee** command is a useful way of recording or diverting some intermediate data in the middle of a pipeline.

Reprise

In this chapter we have noted that the shell interprets certain characters in a special way. We learned about the octothorpe (#), the ampersand (&), the verticule (|), the asterisk (*), the apostrophe ('), the greater-than sign (>), the semicolon (;), and the backslash (\). In a later chapter we'll discuss the remaining special characters: the dollar-sign ($), the apostrophe ('), the less-than sign (<), the question-mark (?), and the double-quote ("),

We have also reviewed the syntax for UNIX commands. Commands can include components such as the *command name*, *options*, *parameters*, *command arguments*, *input files* and *output redirection*.