Chapter 16

The Shell (II)

In Chapter 8 we introduced some of the shell special characters. By way of review, we learned that the shell interprets the octothorpe (#) as the beginning of a comment. By itself, the asterisk (*) is "expanded" by the shell to the names of all files in the current directory. When linked with other characters, such as `A*` or `*B`, the shell expands the expression to the names of all files beginning with A or ending with B. The greather-than sign (>) directs the output to a named file. The vertical bar or pipe (|) allows the output of one command to be directed to the input of the following command. When placed at the end of a command line, the ampersand (&) causes the shell to execute the command as a background process, and immediately returns a prompt so the user can execute other commands. The semicolon (;) indicates the end of a command; this allows more than one command to be placed on a single line. The backslash (\) escapes the special meaning of the immediately following character so it is treated literally. The single quote or apostrophe (') can escape the special meaning of all characters up to the appearance of a matching single quote.

In this chapter we will continue to describe shell special characters and identify their functions. In addition, we will learn about the shell *alias* function.

Shell Special Characters

The remaining special shell characters include the following: the dollars sign ($), the greve (`), the less-than sign (<), the question mark (?), and the double quote ("). We'll consider the function of each of these characters in turn.

Shell Variables

Like any programming language, the shell allows information to be stored and retrieved through shell *variables*. Variables can be given all sorts of names, such as `value`, `Meter`, `A34x` and `BARLINE3`. In order to retrieve information from a variable, the variable name is preceded by a dollars sign. For example, the string `$VARIABLE` means "the current value of the variable named `VARIABLE`. Suppose you had a file named $FILE in the current directory ($FILE is a legitimate filename on UNIX systems). If you type:

```
sort $FILE
```

The shell will assume that there is a variable named FILE, and retrieve its contents. Since the contents are likely to be empty, the above command is identical to typing:

```
sort
```

In order to sort the file named $FILE, the dollars sign would need to be escaped:

```
sort \$FILE
```

Depending on the type of shell, variables can be assigned numerical or string values in various ways. For most shells, variables can be assigned using the equals sign (with no intervening spaces). For example, the integer 7 can be assigned to the variable X as follows:

```
X=7
```

Or the string "hello" can be assigned to a variable by placing the string in quotation marks:

```
X="hello"
```

Single quotation marks can also be used:

```
X='hello'
```

If you had a file named `hello` in the current directory, and if the varialbe `X` had been assigned as above, then the following command would sort this file:

```
sort $X
```

The Shell Greve

It is often useful to be able to save the results of some operation in a shell variable. Suppose for example, that we want to sort a file containing the word `zebra`. But we're not certain what file (or files) contain this word. Manually, we would need to carry out two operations. First we would search for any file(s) containing the word:

```
grep -l zebra *
```

We might find that the word "zebra" appears in the files `animals` and `mammals`. Having determine what files to sort, now we would actually carry out the appropriate sort command:

```
sort animals mammals
```

If we found that word "zebra" occurred in 50 files, then typing the appropriate sort command would require a lot of typing. Alternatively, we could use a shell variable to store the results of the first command, and then retrieve the filenames in the second command. For this, we must use the greve character (`). UNIX shells will execute whatever command(s) appear between two greve characters; the result of the operation can then be treated as a string which may be assigned to a shell variable or used in some other way. In the following commands, the filenames produced by

the **grep** command are assigned to a shell variable named FILES. In the subsequent command a dollars sign instructs the shell to retrieve the contents of this variable:

```
FILES=`grep -l zebra *`
sort $FILES
```

Alternatively, we can avoid the FILES variable altogether, and execute the following command:

```
sort `grep -l zebra *`
```

The shell interprets the above command as follows: First it recognizes the presence of the command delineated by greves. This command is executed *before* the **sort** command. The **grep -l** command will generate as output a string of filenames. This output will replace the material delineated by greves. Finally, the **sort** command will be executed — using the filenames generated by the **grep** command.

This command structure is useful in a variety of circumstances. For example, suppose we wanted to identify any encoded works that are composed by Josquin and are also in triple meter:

```
grep -l '!!!COM: Josquin' `grep -l '!!!AMT:.*triple' *`
```

Here we have imbedded one **grep** "inside" another. Remember that the command delineated by the greve is executed first. In this case, we begin by searching all of the files in the current directory for an AMT reference record containing the keyword "triple." The **-l** option causes the output to consist of only filenames. Then the second **grep** is executed. It looks for files that contain a COM reference record containing the keyword "Josquin." But this second **grep** only searches those filenames passed to it by the first **grep**. In other words, the composer search is restricted to only those files that have a triple meter designation.

Consider another way of using the greve structure. Suppose we have a file named opus16. We would like to know what other works contain the same instrumentation as opus16, but we've forgotten what the precise instrumentation is. We can first seach opus16 for the instrumentation data (encoded in the AIN: reference record), and then search for this information in all files in the current directory. This task can be carried out using a single command line:

```
grep -l `grep '!!!AIN:' opus16` *
```

In this example, the imbedded command provides the regular expression rather than the files to be searched.

Single Quotes, Double Quotes

In Chapter 8 we learned that single quotation marks can be used to escape the special meanings of reserved shell characters — such as * and $. Double quotation marks (") have a similar effect with one important exception. The dollars sign continues to retain its special meaning inside double quotes.

The UNIX **echo** command causes information to be printed or displayed. Consider the following

three commands:

```
echo $A
echo "$A"
echo '$A'
```

In the first and second commands, the shell looks for a variable named A and attempts to echo the contents of this variable on the display. Unless A happens to be a defined shell variable, only an empty line will be displayed. In the third command, the string $A is treated literally, and is echoed back to the display. There are circumstances where the double quotes are more useful, but for most casual users, the single quotes provide the best means for disengaging the meanings of special characters.

Using Shell Variables

Let's consider an example where shell variables prove to be useful in Humdrum processing. Suppose for some score that we want to change the stem-directions in measures 34 through 38 from up-stems to down-stems. First, we need to establish the line number corresponding to the beginning of measure 34 and the line number corresponding to the end of measure 38 (i.e. beginning of measure 39). In the following script, **grep** is used to assign these line numbers to the shell variables $A and $B.

```
A=`grep -n ^=34`
B=`grep -n ^=39`
```

Now we can construct an appropriate **humsed** command. Recall that each substitute (s) command in **humsed** can be preceded by a range indication. In the following command, the $A and $B variables convey the appropriate range to each substitution. This means that the substitutions are limited to the line numbers ranging between $A and $B.

```
humsed "$A,$Bs/\/XXX/g; $A,$Bs//\/g; $A,$Bs/XXX//g" inputfile
```

Notice that we have used double quotes (") rather than single quotes. The quotation marks are necessary to pass all three substitutions as an argument to **humsed**. Using singe quotes, however, would have caused $A and $B to be treated as literal strings rather than shell variables.

Aliases

An alias is an alternative name for something. The shell provides a way of defining aliases, and these aliases can prove very convenient.

Consider, by way of example, the following common pipeline:

```
sort inputfile | uniq -c | sort -n
```

In Chapter 17 we will see that this is a useful way for generating inventories. Typically, this sequence occurs at the end of a pipeline where some preliminary processing has taken place, such

as:

```
timebase -t 8 input | ditto | hint | rid -GLI \
    | sort | uniq -c | sort -n
```

Since the construction `sort | uniq -c | sort -n` is so common, we might want to define an alias for it. To do so, we simply execute the **alias** command. In this case, we've defined a new command called `inventory`:

```
alias inventory="sort | uniq -c | sort -n"
```

Having defined this alias, we can now make use of it. Any time we type the word `inventory`, the shell will expand it to "`sort | uniq -c | sort -n`". The above command can be shortened as follows:

```
timebase -t 8 input | ditto | hint | rid -GLI | inventory
```

Another common task is eliminating barlines. Frequently, we need to use the construction:

```
grep -v ^=
```

Actually, this is not the most prudent construction. Depending on the spines present in a document, sometimes barlines will be mixed with null tokens in other spines that do not encode explicit barlines. E.g.

```
.   =23   =23   .   .   =23
```

A more careful way of eliminating barlines would use the following regular expression:

```
egrep -v '^(\.     )*='
```

That is, eliminate all lines that either begin with an equals-sign, or have one or more leading null tokens followed by a token with a leading equals-sign. Since this is somewhat complicated to remember, we might alias it. In the following command, we have created a new command called `nobarlines`:

```
alias nobarlines='egrep -v '^(\.     )*='
```

In Humdrum, a good use of aliases is to define commonly used regular expressions. Consider the regular expression used to define tandem interpretations that encode meter signatures. Here we are searching for an asterisk at the beginning of a line, followed by the upper-case letter 'M' followed by a digit, followed by zero or more digits, followed by a slash, followed by a digit:

```
grep '^\*M[0-9][0-9]*/[0-9]' inputfile
```

Actually, this regular expression will fail to find any meter signature that is not in the first spine. A more circumspect regular expression will include the possibility of a leading tab:

```
grep '   *\*M[0-9][0-9]*/[0-9]' inputfile
```

Since this is a cumbersome regular expression, it can help to provide an alias. Here we have aliased the regular expression to the name `metersig`:

```
alias metersig="' *\*M[0-9][0-9]*/[0-9]'"
```

Now we can search for meter signatures as follows:

```
grep metersig inputfile
```

Reprise

In this chapter we have discussed how the shell interprets the dollars sign ($), the greve (‘), and the double quote ("). When followed by printable characters, the dollars sign is interpreted as designating the value of a shell variable. Any command enclosed between two greve characters is executed by the shell first, and the returned output of the command is available as an input parameter to some other command. Like single quotes, double quotes can be used to escape special shell characters; however, an important difference is that the dollars-sign retains its special meaning within the double quotes. This allows shell variables to be embedded into text strings.

We have also learned that the shell **alias** command can be used to provide a convenient short-hand or way of abbreviating a complex pipeline or regular expression into a single user-defined keyword.