Chapter 32

The Shell (IV)

In research applications, it is impossible to anticipate all the types of manipulations we might want to carry out. For some tasks, we will need to write our own software to carry out specific operations of interest. Fortunately, many specialized tasks require only a brief program to achieve the goal. The Humdrum tools can be used in conjunction with user-developed software to carry out specific tasks.

Many users will already have some programming ability and will be able to apply this knowledge using their preferred programming language. For those users who have less programming background, it may be useful to learn some basic programming skills. While the shell provides a useful programming environment, for more complex tasks, it is better to use one of the many good programming languages.

For data manipulation tasks comparable to those described in this book, the most appropriate programming language include **perl** and **awk**. The **awk** programming language is especially useful for text processing, retrieving, transforming, reducing, and validating text data. The **perl** programming language provides even more extensive capabilities, but requires a somewhat greater effort to learn. For research-oriented programming, **perl** is the programming language of choice. However, for this brief introduction we will describe features of the **awk** programming language. Awk is a so-called "scripted" language. It is easy to learn but nevertheless quite powerful.

The *awk* Programming Language

Awk programs can be executed from the shell command line. A simple program is the following:

```
awk '{print "hello"}'
```

The **awk** command invokes the awk program interpreter. The material within the single quotes is the actual program. Once the program is started, it is is executed once each time you type the carriage return or ENTER key. To stop the program, simply type control-D (on UNIX systems) or control-Z (on DOS systems).

In the default configuration, an awk program will be executed once for each line of input. If no input file is specified, then "standard input" is assumed. That is, input will come from either data arriving through a pipeline, or data typed at the keyboard.

Automatic Parsing of Input Data

Each line of input data is automatically assigned to the awk variable `$0`. This means that the command

```
awk '{print $0}'
```

will simply echoe each line of input as the output. Similarly, the following command will print each line of input preceded by a colon and a space:

```
awk '{print ": " $0}'
```

For any input line, awk also automatically parses the data into individual tokens or fields. A token is deemed to be any sequence of characters that is separated from other tokens by any blank space such as spaces or tabs. The first data token is automatically assigned to an awk variable `$1`. The second data token is assigned to the variable `$2`, and so on. For example, suppose a program encountered the following input line:

```
243xyz 3    29    #%$    **      Ullyses 234-034
```

The variables would be automatically assigned as follows:

```
$1 = 234xyz
$2 = 3
$2 = 29
$4 = #%$
$5 = **
$6 = Ullyses
$7 = 234-034
```

Given this input, the command

```
awk '{print $2 + $3}'
```

will print the sum of `$2` and `$3`, namely 32.

Arithmetic Operations

Suppose that we have two `**semits` spines as input and we would like to print the semitone difference between the two parts for each sonority. Typically, the higher part is placed in the rightmost spine, so it makes most sense to subtract $1 from $2. Negative numbers mean that the nominally lower part has crossed above the nominally higher part:

```
awk '{print $1 - $2}'
```

In addition to addition and subtraction, other possible arithemtic operators include the slash (/) for division, the asterisk (*) for multiplication, the caret (^) for exponentiation, and the percent sign (%) for modulo arithmetic. Parentheses can be used to clarify the order of operations. For exam-

ple, the following command prints the product of the first and second tokens ($1 * $2) divided by the third token raised to the fourth token power:

```
awk '{print ($1 * $2) / ($3^$4)}'
```

As we have already seen, character strings can also be included in print statements. For example, we might want to print the first and third input tokens separated by a tab:

```
awk '{print $1 "\t" $3}'
```

Conditional Statements

Often we'd like to avoid processing certain records. For example, we might wish to avoid processing barlines. The awk **if** statement can be used to restrict the operation to particular circumstances. Consider the following awk program:

```
awk '{if ($0 !~/^=/) print $1 - $2}'
```

The **if** condition is given in parentheses. The string given between the slashes (`/^=/`) is a regular expression: in this case, it identifies any equals sign that occurs at the beginning of an input line. The tilde means "match" and the exclamation mark means "not". Hence the program means: if the entire line (`$0`) does not match (`!~`) an equals sign occuring at the beginning of the line (`/^=/`), then print the value of the first token minus the value of the second token (`print $1 - $2`).

Awk also provide an **else** condition. The syntax is:

```
if (condition)
    [then] {do something}
    else {do something else instead}
```

For Humdrum inputs, we may want to avoid processing comments and interpretations. Whenever we encounter a comment or interpretation, we might simply echo the input record in the output:

```
awk '{if($0 ~/^[*!]/) {print $0} else {print $1 - $2}}'
```

Sometimes we might simply want to do nothing at all when we encounter a comment or interpretation:

```
awk '{if($0 ~/^[*!]/) {} else {print $1 - $2}}'
```

Recall that input tokens in awk are separated by any blank space such as spaces or tabs. This means that a Humdrum multiple-stop will be treated as containing two or more tokens. We can avoid this situation by explicitly telling awk to assign the "field separate" (FS) to the tab character. For example, the following program prints the value in the third spine of a Humdrum input. Without reassigning the field separator, the third token might be the third element of a multiple-stop in the first spine, or the second element of a multiple-stop appearing in the second spine.

```
awk '{FS="\t"; print $3}'
```

Notice the use of the semicolon to separate individual instructions.

Assigning Variables

Within an awk program, the user can assign and manipulate variables that store particular values. Variables may hold numerical values or they may hold character strings. In the following examples, the value 178 is assigned to the variable 'A'; the value 2.2 is assigned to the variable 'number'; and the character string "Dear Gail" is assigned to the variable 'salutation':

```
A=178
number = 2.2
salutation = "Dear Gail"
```

Named variables can be used for various arithmetic operations. For example:

```
A=178+18
number = 2.2 + A
number_squared = number ^ 2
```

Manipulating Character Strings

Variables holding character strings can be concatenated together. In the following example, after the first three assignments, the variable saluation will contain the character string "Dear Craig":

```
opening = "Dear"
space = " "
name = "Craig"
salutation = opening space name
```

Awk provides a number of built-in functions for manipulating text. One function (**gsub**) carries out global substitutions. The syntax is:

```
gsub("target-string","replacement-string",variable)
```

For example, the following instruction changes all occurrences of X to Y in a variable named string:

```
gsub("X","Y",string)
```

Suppose that we wanted to increment all measure numbers by 1. Let's presume our input contains only a single spine. First we test for the presence of the equal sign at the beginning of the input record. If the input is not a barline, then we simple print the line in the output. Otherwise we: (1) assign the input to the variable barline, (2) eliminate all non-numeric characters using gsub, (3) add one to the remaining numeric value, and (4) output the new number preceded by the equal

sign:

```
awk '{
     if ($0 !~/^=/) {print $0}
     else {
          barline = $1
          gsub("[^0-9]","",barline)
          barline = barline + 1
          print "=" barline
          }
     }'
```

Notice that we are at liberty to add spaces, tabs, and newlines in order to improve the readability of our program.

The *for* Loop

Often we would like to repeat a process for several concurrent spines. For example, suppose we had four spines of **solfa data and we want to output the total number of leading-tones for each sonority. Awk provides a **for** instruction that allows us to cycle through a series of values. The **for**-loop construction has the following syntax:

```
for (initial-value; condition-for-continuing; increment-action)
     {do something repeatedly}
```

In the case of counting the number of leading-tones for each of four spines, our program would be as follows:

```
awk '{
     count = 0
     for (i=1; i<=4; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

The initial value for the for-loop is 1 (`i=1`); each time the loop is executed the value of `i` is incremented by 1 (`i++`); and the loop continues executing as long as `i` is less-than or equal to 4 (`i<=4`). The value `$i` will take successive values so that the loop will test whether each of $1, $2, $3 and $4 match the regular expression `/ti/`. For each match, the variable `count` is incremented by 1. Finally, the value of count is printed. The count is set to zero each time the program is run (that is, for each line of input).

It would be nice if our program could adapt to inputs containing any number of spines. For each line of input, awk automatically identifies the number of input tokens or fields and stores the value in the varible `NF`. Simply replacing the number 4 by `NF` will achieve our goal. In our revised program we have also added some comments to clarify our code. Like the shell, awk comments consist of material following the octothorpe character (#):

```
awk '{
     # A program to count occurrences of the leading-tone.
     count = 0
     for (i=1; i<=NF; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

A problem with the above script is that it will attempt to count occurrences of `ti` in Humdrum comments, interpretations, and barlines. We can improve our program by echoing these in the output without processing them. Another refinement makes use of the awk **next** instruction. Whenever a **next** statement is encountered, the program immediately moves on to the next input line and begins processing again from the start of the program.

```
awk '{
     # A program to count occurrences of the leading-tone.
     count = 0
     if ($0 ~/^[!*=]/) {print $0; next}
     for (i=1; i<=NF; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

Although our output data will consist of a single column (spine) of numbers, it is possible that an input will contain more than one interpretation — and so cause the output to fail to conform to the Humdrum syntax. Rather than simply echoing any interpretation records, we might ensure that only a single interpretation is generated for the output. First, we might look for exclusive interpretations (beginning `**`) and output a suitable interpretation of our own (e.g., `**leading-tones`). In the case of tandem interpretations (beginning with only a single asterisk), we could output a single null interpretation. Similarly, when we encounter a barline, we might ensure that only one barline token is output. Finally, we should remain vigilant for spine-path terminators (`*-`) and ensure that our output is similarly properly terminated. The revised program is as follows:

```
awk '{
     # A program to count occurrences of the leading-tone.
     count = 0
     if ($0 ~/^**/) {print "**leading-tones"; next}
     if ($0 ~/^*-/) {print "*-"; next}
     if ($0 ~/^*[^*]/) {print "*"; next}
     if ($0 ~/^!!/) {print $0; next}
     if ($0 ~/^=/) {print $1; next}
     {
     for (i=1; i<=NF; i++)
          {if ($i ~/ti/) count++}
     print count
}'
```

Of course there are many other features of the awk programming language that we have not de-

scribed here. These features include associative arrays, built-in variables, string-processing functions, user-defined functions, system calls, begin and end blocks, other control-flow statements, and pipes and file manipulations.

Reprise

In this chapter we have introduced some features of the **awk** pattern/action language. A programming language, like **awk** or **perl** can be used to transform data in highly specific and specialized ways. The power of Humdrum is significantly enhanced when users are able to create their own specialized filters.