Chapter 24

The Shell (III)

In Chapter 16 we learned about the **alias** feature of the shell. The **alias** command allowed us to create new commands by assigning a complex pipeline to a single-word command. In this chapter we will learn how to use the shell to write more complex programs. Shell programs allow users to reduce lengthy sequences of Humdrum commands to a single user-defined command.

Shell Programs

A shell program is simply a script consisting of one or more shell commands. Suppose we had a complex procedure consisting of a number of commands and pipelines:

```
extract -i '**Ursatz' inputfile | humsed '/X/d' \
    | context -o Y -b Z > Ursatz
extract -i '**Urlinie' inputfile | humsed '/X/d' \
    | context -o Y -b Z > Urlinie
assemble Ursatz Urlinie | rid -GLId | graph
```

In the above hypothetical script, we have processed an input file called `inputfile`. It may be that this is a procedure we would like to apply to several different files. Rather than typing the above command sequence for each file, an alternative is to place the above commands in a file. Let's assume that we put the above commands in a file called `Schenker`. In order to execute this file as a shell script, we need to assign *execute permissions* to the file. We can do this by invoking the UNIX **chmod** command.

```
chmod +x Schenker
```

The **+x** option causes **chmod** to add execute permissions to the file `Schenker`. Using **chmod** we can change modes related to *executing* a file, *reading* a file, and *writing* to a file. Possible mode changes include the following:

+x	add execute permission
-x	deny execute permission
+r	add read permission
-r	deny read permission

+w	add write permission
-w	deny write permission

Having added execute permissions to the file, we can now execute the shell script or program. This is done simply by typing the name of the file; in effect, the filename becomes a new command:

```
Schenker
```

Each time we type this command, our script will be executed anew. Notice that in our script, the final output has not been sent to a file. As a result, the output from our **Schenker** command will be sent to the screen (standard output). It is convenient not to specify an output file in the script since this is often something the user would like to specify. When typing our new command, we can use file-redirection to place the output in a user-specified file:

```
Schenker > outputfile
```

As currently written, our program can be applied only to an input file whose name is literally `inputfile`. If we wanted to, we could edit our script and up-date the name of the input filename every time we want to use the command. However, it would be more convenient to specify the input filename on the command line — as we can do for other commands. For example, it would be convenient to be able to type commands such as the following:

```
Schenker opus118 > opus118.out
```

In order to allow such a possibility, we can use a predefined feature of the shell. Whenever the shell receives a command, each item of information on the command line is assigned to a shell variable. The first item on the command line is assigned to the variable $0 (normally, this is the command name). For example, in the above example, $0 is assigned the string value "`Schenker`." The variables $1, $2, $3, etc. are assigned to each successive item of information on the command line. So in the above example, $1 is assigned the string value "`opus118`."

These shell variables can be accessed within the shell script itself. We need to revise the script so that each occurrence of the input file is replaced by the variable $1:

```
extract -i '**Ursatz' $1 | humsed '/X/d' \
   | context -o Y -b Z > Ursatz
extract -i '**Urlinie' $1 | humsed '/X/d' \
   | context -o Y -b Z > Urlinie
assemble Ursatz Urlinie | rid -GLId | graph
```

This change means that our **Schenker** command can be applied to any user-specified input file — simply by typing the filename in the command.

Flow of Control: The *if* Statement

Suppose we wanted our **Schenker** command to apply only to tonal works — more specifically, to works with a known key. Before processing a work, we might want to have **Schenker** test for the presence of a tandem interpretation specifying the key.

Let's begin by using **grep** to search for a key tandem interpretation. An appropriate **grep** command would be:

```
grep '^\*[A-Ga-g][#-]*:' $1
```

Recall that we can assign the output of any command to a shell variable by placing the command within back-quotes or greves, i.e. '...'. Let's assign the key interpretation to the variable KEY:

```
KEY=`grep '^\*[A-Ga-g][#-]*:' $1`
```

If no key indicator is found by **grep**, then the variable KEY will be empty. We can test for this condition using the shell **if** statement.

```
KEY=`grep '^\*[A-Ga-g][#-]*:' $1`
if [ "$KEY" = "" ]
then
          echo "Sorry, this input file has no key."
          exit
fi
```

Notice that we use the dollars sign prior to the variable to mean "the contents of variable KEY". The double quotation marks allow a string comparison. Our test is whether the variable $KEY is equivalent to the empty or null string "". If the test is true, then the commands following the `then` statement are executed. By convention, these commands are indented for clarity. In the above case, two commands are executed if the $KEY variable is empty. The **echo** command causes the quoted string to be output. The **exit** command causes the script to terminate. Notice the presence of the **fi** command (**if** backwards). This command simply indicates that the if-block has ended.

Of course, if there is a key designation, then it is appropriate to execute the rest of our **Schenker** script. The complete script would be as follows:

```
KEY=`grep '^\*[A-Ga-g][#-]*:' $1`
if [ "$KEY" = "" ]
then
          echo "Sorry, this input file has no key."
          exit
else
          extract -i '**Ursatz' $1 | humsed '/X/d' \
             | context -o Y -b Z > Ursatz
          extract -i '**Urlinie' $1 | humsed '/X/d' \
             | context -o Y -b Z > Urlinie
          assemble Ursatz Urlinie | rid -GLId | graph
fi
```

Notice the addition of the **else** statement. The **else** statement delineates the block of commands to be executed whenever the **if** condition fails — that is, when the $KEY variable does *not* equal the null string. Once again, to make the script more readable, we indent the commands contained in the else-block.

The **if** command provides many other ways of testing some condition. For example, the shell provides ways to determine whether a file exists, and other features.

Flow of Control: The *for* Statement

In music research, a common task is to apply a particular process or script to a large number of score files. By way of illustration, suppose we wanted to know the maximum number of notes in any single folk melody in a collection of Czech folksongs. Suppose further that we are located in a directory containing a large number of Czech folksongs named `czech01.krn`, `czech02.krn`, `czech03.krn`, and so on.

We would like to run the **census -k** command on each file separately, but we'd prefer not to type the command for each score. The **for** statement provides a convenient way to do this. The following commands might be typed directly at the shell:

```
for J in czech*.krn
> do
> census -k $J | grep 'Number of notes:'
> done | sort -n
```

The pattern `czech*.krn` will be expanded to all of the files in the current directory that it matches. The variable **J** will take on each name in turn. The commands between **do** and **done** will be executed for each value of the variable **$J**. That is, initially **$J** will have the value `czech01.krn`. Having completed the do-done block of commands, the value of **$J** will become `czech02.krn`, and the do-done block will be repeated. This will continue until the value of **$J** has taken on all of teh possible matches for `czech*.krn`.

The output might appear as follows:

```
Number of notes:            31
Number of notes:            32
Number of notes:            32
Number of notes:            34
Number of notes:            35
Number of notes:            39
Number of notes:            39
Number of notes:            40
Number of notes:            48
Number of notes:            48
Number of notes:            55
Number of notes:            78
etc.
```

Incidentally, the output from a **for** construction such as above can be piped to further commands, so we might identify the maximum number of notes in a Czech melody by piping the output through **sort -n**.

A Script for Identifying Transgressions of Voice-Leading

Shell programs can be of arbitrary complexity. Below is a shell program (dubbed **leader**) whose purpose is to identify all instances of betrayals of nine classic rules of voice-leading for a two-part input. A number of refinements have been added to the program — including input file checking, and formatting of the output.

The program is invoked as follows:

```
leader <file>
```

The input is assumed to contain two voices, each in a separate `**kern` spine. The nominally lower voice should be in the first spine. For music containing more than two voices, the Humdrum **extract** command should be used to select successive pairs of voices for processing by **leader**.

```
# LEADER
#
# A shell program to check for voice-leading infractions.
# This command is invoked as:
#
#   leader <filename>
#
# where <filename> is assumed to be a file containing two voices, each
# in a separate **kern spine, where the nominally lower voice is in the
# first spine.

# Before processing, ensure that a proper input file has been specified.
if [ ! -f $1 ]
then   echo "leader: file $1 not found"
       exit
fi
if [ $# -eq 0 ]
then   echo "leader: input file not specified"
       exit
fi

# 1. Record the ranges for the two voices.
echo 'Range for Upper voice:'
extract -f 2 $1 | census -k | egrep 'Highest|Lowest' | sed 's/^/      /'
echo 'Range for Lower voice:'
extract -f 1 $1 | census -k | egrep 'Highest|Lowest' | sed 's/^/      /'

# 2. Check for augmented or diminished melodic intervals.
extract -f 1 $1 | mint -b r | sed '/\[[Ad][Ad]*\]/d' | egrep -n '^[^!*].*[Ad][^1]' |\
    sed 's/:/ (/;s/$/)/;s/^/Augmented or diminished melodic interval at line: /'
extract -f 2 $1 | mint -b r | sed '/\[[Ad][Ad]*\]/d' | egrep -n '^[^!*].*[Ad][^1]' |\
    sed 's/:/ (/;s/$/)/;s/^/Augmented or diminished melodic interval at line: /'

# 3. Check for consecutive fifths and octaves.
```

```
echo 'P5'  > $TMPDIR/template;  echo 'P5'  >> $TMPDIR/template
hint -c $1 | patt -f $TMPDIR/template -s = | \
    sed 's/ of file.*/./;s/.*Pattern/Consecutive fifth/'
echo 'P1'  > $TMPDIR/template;  echo 'P1'  >> $TMPDIR/template
hint -c $1 | patt -f $TMPDIR/template -s = | \
    sed 's/ of file.*/./;s/.*Pattern/Consecutive octave/'

# 4. Check for doubling of the leading-tone.
deg $1 | extract -i '**deg' | ditto -s = | sed 's/^=.*/=/' | \
    egrep -n '^7.*7|^[^!*].*7.*7' | egrep -v '7[-+]' | \
    sed 's/:.*/./;s/^/Leading-tone doubled at line: /'

# 5. Check for unisons.
semits -x $1 | ditto -s = | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1==$2) print "Unison at line: " NR}'

# 6. Check for the crossing of parts.
semits -x $1 | ditto -s = | sed 's/^=.*/=/' | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1>$2) print "Crossed parts at line: " NR}'

# 7. Check for more than an octave between the two parts.
semits -x $1 | ditto -s = | awk '{if($0~/[^0-9\t-]/)next} \
    {if($2-$1>12) print "More than an octave between parts at line: " NR}'

# 8. Check for overlapping parts.
extract -f 2 $1 | sed 's/^=.*/./' | context -n 2 -p 1 -d XXX | \
    rid -GL | humsed 's/XXX.*//' > $TMPDIR/upper
extract -f 1 $1 | sed 's/^=.*/./' > $TMPDIR/lower
assemble $TMPDIR/lower $TMPDIR/upper | semits -x | ditto | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1>$2) print "Parts overlap at line: " NR}'
extract -f 1 $1 | sed 's/^=.*/./' | context -n 2 -p 1 -d XXX | \
    rid -GL | humsed 's/XXX.*//' > $TMPDIR/lower
extract -f 2 $1 | sed 's/^=.*/./' > $TMPDIR/upper
assemble $TMPDIR/lower $TMPDIR/upper | semits -x | ditto | \
    awk '{if($0~/[^0-9\t-]/)next}{if($1>$2) print "Parts overlap at line: " NR}'

# 9. Check for exposed octaves.
hint -c $1 > $TMPDIR/s1
extract -f 1 $1 | deg > $TMPDIR/s2
extract -f 2 $1 | deg > $TMPDIR/s3
extract -f 1 $1 | mint | humsed 's/.*[3-9].*/leap/' > $TMPDIR/s4
extract -f 2 $1 | mint | humsed 's/.*[3-9].*/leap/' > $TMPDIR/s5
assemble $TMPDIR/s1 $TMPDIR/s2 $TMPDIR/s3 $TMPDIR/s4 $TMPDIR/s5 > $TMPDIR/temp
egrep -n 'P1.*\^.*\^.*leap.*leap|P1.*v.*v.*leap.*leap' $TMPDIR/temp | \
    sed 's/:.*/./;s/^/Exposed octave at line: /'

# Clean-up some temporary files.
rm $TMPDIR/template $TMPDIR/upper $TMPDIR/lower $TMPDIR/s[1-5] $TMPDIR/temp
```

Reprise

In this chapter we have illustrated how to package complex Humdrum command scripts into shell programs. This allows us to create special-purpose commands. We learned that files can be transformed into executable scripts through the **chmod** command. We also learned how to pass parameters from the command line to the script, and how to assign and modify the contents of variables. In addition, we learned how to influence the flow of control using the **if** and **for** statements. Finally, we learned that multi-line scripts can be typed directly at the command line without creating a script file.

Shell scripts can be very brief or very long. It is possible to create scripts that carry out highly sophisticated processing such as searching for voice-leading transgressions. There are innumerable features to shell programming that have not been touched-on in this chapter. Several books are available that provide comprehensive tutorials for shell programming.