Chapter 10

Musical Uses of Regular Expressions

Now that you have a better understanding of regular expressions, let's apply them. This chapter provides many examples of how regular expressions may be used to define musically useful patterns. In subsequent chapters, we'll make frequent use of regular expressions.

The *grep* Command (Again)

Although regular expressions are used in a number of Humdrum commands, they are most frequently used in conjunction with the **grep** command encountered in Chapter 3. **grep** is a popular software tool that is available from a number of manufacturers and sources. Many versions of **grep** differ in the options provided. For example, the version of **grep** distributed by the GNU Software Foundation provides no fewer than 19 options. Some of the most common options for **grep** are identified in Table 10.1.

Table 10.1

-c	count the number of lines matching the regular expression
-f *file*	search for patterns that are specified in *file*
-i	ignore differences of upper- and lower-case
-l	just list the names of files containing a matching line
-n	prefix each output line with its line number
-h	suppress file-name prefixes (headers) in output when searching more than one file
-v	display all lines *not* matching the regular expression
-L	list names of files *not* containing the regular expression

*Common options for the **grep** command.*

Many of the predefined Humdrum representations make use of the "common system" for representing barlines. The following command counts the number of barlines in the file `czech37.krn`. Note that the caret anchor (`^`) is used to avoid inadvertent matches of the equals sign that might appear in Humdrum comments or interpretations.

```
grep -c ^= czech37.krn
```

Recall that the dollar sign (`$`) can be used to anchor an expression to the end of the line. The following command determines whether numbered measure 9 is present in the file `france12.krn`; the dollar sign ensures that measure 9 is not mistaken for measure 90, 930, etc.

```
grep ^=9$ france12.krn
```

The asterisk means "zero or more" instances of the preceding expression. For example, the following regular expression will match any reference record or global comment in the file `clara29`:

```
grep '^!!!*' clara29
```

Suppose we want to list all of the global comments for all files in the current directory:

```
grep '^!!!*' *
```

Notice that the two asterisks serve different functions in the above command. The first asterisk means "zero or more instances" and is part of the regular expression passed to **grep**. The second asterisk means "all files in the current directory" and is expanded by the shell. The first asterisk is 'protected' from the shell by the single quotes. Otherwise, the first asterisk might be expanded by the shell to a list of all files in the current directory.

In regular expressions, the period character (`.`) matches any single character. For example, the expression '`A.B`' will match strings such as '`AXB`' and '`AAB`' etc. The following command identifies all eighth-notes containing at least one flat, and whose pitch lies within an octave of middle C.

```
grep 8.- *.krn
```

Frequently it is necessary to turn off the special meanings for metacharacters such as ^, $, and *. Recall that this can be done by inserting a backslash (`\`) immediately prior to the metacharacter. In the `**kern` representation the caret signifies an accent. In a monophonic input, we might count the number of notes that have a notated accent as follows:

```
grep -c '\^' danmark3.krn
```

In the following command we have used the backslash to escape the special meaning of the asterisk. The **-l** option causes **grep** to output only the names of any files that contain a line matching the pattern. Hence, the following command identifies those files in the current directory that encode music in 9/8 meter:

```
grep -l '^\*M9/8' *
```

Recall that square brackets can be used to indicate character classes where any of the characters in the class can be used to match the expression. The following command identifies those files in the current directory that encode music in either 3/8 or 9/8 meter:

```
grep -l '\*M[39]/8' *
```

One of the most frequently used regular expressions consists of the period followed by the asterisk (`.*`). Recall that this expression will match *any* string including the null string (i.e. nothing at all). This expression commonly appears between two other character strings. For example, we can identify all files in the current directory whose instrumentation includes a trumpet:

```
grep -l '!!!AIN.*tromp' *
```

The `.*` expression is needed since we don't know what other instruments might be listed following `AIN` and before `tromp`. Instrumentation reference records require that instrument codes appear in alphabetical order. This makes it easier to conduct searches for combinations of instruments. For example, we can identify all scores in the current directory whose instrumentation includes both trumpet and cornet as follows:

```
grep -l '!!!AIN.*cornt.*tromp' *
```

There are many variants on the use of the `.*` expression. The following command identifies all files that contain a record having the word `Drei` followed by the word "`Koenige`". (Notice the use of the **-i** option in order to ignore the case of the letters.)

```
grep -li 'Drei.*Koenige' *
```

This command will match such strings as: *Die Heiligen Drei Koenige*, *Drei Koenige, Dreikoenigslied,* etc.

The '`!!!AGN`' reference record is used to encode genre-related keywords. The following command lists all files that are ballads.

```
grep -l '!!!AGN.*Ballad' *
```

List all files that have the word `Amour` in the title:

```
grep -li '!!!OLT.*Amour' *
```

List any works that bear a dedication:

```
grep -l '!!!ODE:' *
```

List those works that are in irregular meters:

```
grep -l '!!!AMT.*irregular' *
```

The **-L** option for **grep** causes the output to contain a list of files *not* containing the regular expression. For example, we could identify those works that don't bear any dedication:

```
grep -L '!!!ODE:' *
```

List those works *not* composed by Schumann:

```
grep -L '!!!COM: Schumann' *
```

Identify any works that don't contain any double barlines:

```
grep -L '^==' *
```

How many works in the current directory are in simple-triple meter?

```
grep -c '!!!AMT.*simple.*triple' *
```

When searching for more complex patterns it may be necessary to use **grep** more than once. Consider, for example, the problem of identifying works whose titles contain both the words `Liebe` and `Tod`. The first of the following commands will identify only those titles that contain `Liebe` followed by `Tod`, whereas the second command will identify only those titles that contain `Tod` followed by `Liebe`:

```
grep '!!!OTL.*Liebe.*Tod' *
grep '!!!OTL.*Tod.*Liebe' *
```

A better solution is to pipe the output between two **grep** commands. Recall that the vertical bar ('|') conveyes or "pipes" the output from one command to the input of a subsequent command. The following command passes all opus-title records (`OTL`) containing the word `Liebe` to a second **grep**, which passes only those records also containing the word `Tod`. Since both **grep** commands process the entire input line, it does not matter whether the word `Tod` precedes or follows the word `Liebe`:

```
grep '!!!OTL.*Liebe' * | grep 'Tod'
```

The **-v** option for **grep** causes a "reverse" or "negative" output. Instead of outputting all records that *match* the specified regular expression, the **-v** option causes only those records to be output that do *not* match the given regular expression. For example, the following command eliminates all comments from the file `polska24.krn`:

```
grep -v '^!' polska24.krn
```

Similarly, the following command eliminates all whole-note rests:

```
grep -v 1r *
```

The **-v** option is especially convenient in pipelines. For example, the following command identifies all those files whose instrumentation includes a cornet but not a trumpet:

```
grep '!!!AIN.*cornt' * | grep -v 'tromp'
```

The following command identifies those works in compound meters that are not also quadruple meters:

```
grep '!!!AMT.*compound' * | grep -v 'quadruple'
```

Similarly, the following command identifies those notes that begin a phrase, but are not rests.

```
grep '^{' * | grep -v r
```

German, French, Italian, and Neapolitan Sixths

In conjunction with the **solfa** command, **grep** can be used to search for various types of special chords. Suppose, for example, that we wanted to identify occurrences of augmented sixth chords. An augmented sixth chord is characterized by an augmented sixth interval occurring between the lowered sixth scale-degree and the raised fourth scale-degree. In Chapter 4, we saw that the **solfa** command represents pitches with respect to an encoded tonic pitch. In the `**solfa` representation, the lowered sixth and raised fourth degrees will be represented as `6-` and `4+` respectively. First we translate the input to the `**solfa` representation, and then we search for records matching the appropriate regular expression:

```
solfa input | grep '6-.*4+'
```

Notice that the expression '`6-.*4+`' presumes that the lowered sixth degree is lower in pitch than the raised fourth degree. For augmented sixth chords, this is a reasonable presumption. In the unlikely situation that the raised fourth degree is lower in pitch than the lowered sixth degree, we would need to also search for the expression '`4+.*6-`'. Alternatively, we could use two separate **grep** commands, eliminating the constraint of order:

```
solfa input | grep '6-' | grep '4+'
```

Augmented sixth chords can be further classified as either German, French, or Italian sixths. The German sixth contains the lowered mediant whereas the French sixth contains the supertonic pitch; the Italian sixth contains neither:

```
solfa input | grep '6-.*4+' | grep '3-'          # German sixth
solfa input | grep '6-.*4+' | grep '2'           # French sixth
solfa input | grep '6-.*4+' | grep -v '[23]'     # Italian sixth
```

A similar approach can be used to identify Neapolitan sixth chords. These chords are based on the lowered supertonic with the third of the chord (unaltered subdominant) in the bass.

```
solfa input | grep '4[^-+].*2-' | grep '6-'      # Neapolitan sixth
```

Depending on the key, Neopolitan chords are sometimes notated enharmonically as a raised tonic chord. Suppose we were looking for such enharmonically spelled Neopolitan chords:

```
solfa input | grep '3+.*1+' | grep '5+'
```

Occassionally, Neapolitan chords are missing the fifth of the chord (the lowered sixth degree of the scale). We might search for an example of such a chord:

```
solfa input | grep '2-' | grep '4' | grep -v '6-'
```

AND-Searches Using the *xargs* Command

In some cases, we want to identify those files that match two entirely different patterns (in different records). Recall that the **-l** option causes **grep** to output the *filename* rather than the matching

record. If we could pass along these file names to another **grep** command, we could search those same files for yet another pattern.

The UNIX **xargs** command provides a useful way of transferring the output from one command to be used as final arguments for a subsequent command. For example, the following command takes each file whose opus title contains the word `Liebe` and counts the number of phrases.

```
grep -l '!!!OTL:.*Liebe' * | xargs grep -c '^{'
```

In this case the **grep -l** command outputs a list of names of files containing the string `Liebe` in an OTL reference record. The **xargs** command causes these filenames to be appended to the end of the following **grep** command. The **grep -c** command will thus be applied only to those files already identified by the previous **grep** as containing `Liebe` in the title.

A set of such pipelines can be used to answer more sophisticated questions. For example, are drinking songs more apt to be in triple meter?

```
grep -l '!!!AMT.*triple'  *   | xargs grep -l '!!!AGN.*Trinklied'
grep -l '!!!AMT.*duple'   *   | xargs grep -l '!!!AGN.*Trinklied'
grep -l '!!!AMT.*quadruple' * | xargs grep -l '!!!AGN.*Trinklied'
```

Similarly, the following commands determine whether files whose titles contain the word *death* are more apt to be in minor keys:

```
grep -li '!!!OTL.*death' * | xargs grep -c '^\*[a-g][#-]*:'
grep -li '!!!OTL.*death' * | xargs grep -c '^\*[A-G][#-]*:'
```

Note that the **xargs** command can be used again and again to continue propagating file names as arguments to subsequent searches. For example, the following command outputs the key signatures for all works originating from Africa that are written in 3/4 meter:

```
grep -l '!!!ARE.*Africa' * | xargs grep -l '^\*M3/4' \
    | xargs grep '^\*k\['
```

Similarly, the following command outputs the names of all files in the current directory that encode 17th century organ works containing passages in 6/8 meter:

```
grep -l '!!!ODT.*16[0-9][0-9]/' | xargs grep -l \
     '!!!AIN.*organ' | xargs grep -l '\*M6/8'
```

Using the **-L** option allows us to form even more complex criteria by excluding certain works. For example, the following variation of the above command outputs the names of all files in the current directory that encode 17th century organ works that do not contain passages in 6/8 meter:

```
grep -l '!!!ODT.*16[0-9][0-9]/' | xargs grep -l \
     '!!!AIN.*organ' | xargs grep -L '\*M6/8'
```

OR-Searches Using the *grep -f* Command

In effect, the above pipelines provide logical **AND** structures: e.g. identify works composed in the 17th century AND written for organ AND containing a passage in 6/8 meter. The **-f** option for **grep** provides a way of creating logical **OR** searches. With the **-f** option, we specify a file containing the patterns being sought. For example, we might create a file called `criteria` containing the following three regular expressions:

```
!!!ODT.*16[0-9][0-9]/
!!!AIN.*organ
\*M6/8
```

We would invoke **grep** as follows:

```
grep -l -f criteria *
```

The **-f** option tells **grep** to fetch the file `criteria` and use the records in this file as regular expressions. A match is made if any of the regular expressions is found.

The output would consist of a list of all files in the current directory that encode works composed in the 17th century OR written for organ OR in 6/8 meter. The **-f** option is more typically used to specify several variations of the same idea. For example, suppose we were searching for D major triads in `**pitch` data. We could use a file containing the following regular expressions:

```
[Dd].*[Ff]#.*[Aa]
[Dd].*[Aa].* [Ff]#
[Ff]#.*[Aa].*[Dd]
[Ff]#.*[Dd].*[Aa]
[Aa].*[Dd].*[Ff]#
[Aa].*[Ff]#.*[Dd]
```

Depending on the application, it may be easier to construct such pattern files than to use a lengthy pipeline. That is:

```
grep -f Dmajor *
```

may be less cumbersome than:

```
grep [Dd] * | grep [Ff]# | grep [Aa]
```

The **-f** option can be combined with **-L**. For example, suppose we wanted to identify all works in the current directory that are not in the keys of C major, G major, B-flat major or D minor. Our regular expression file would contain the following regular expressions:

```
^\*[CGd]:
^\*B-:
```

The corresponding command would be:

```
grep -L -f criteria *
```

Another way of thinking of the **-f** option is that it allows us to define equivalences. Consider, for example, the task of counting all of the notes in a `**kern` melody that belong to a particular whole-tone pitch set. Let's create two files, one called `whole1` and the other called `whole2`. The file `whole1` might contain the following regular expressions:

```
[Cc]([^-#Cc]|$)
[Dd]([^-#Dd]|$)
[Ee]([^-#Ee]|$)
[Ff]#([^#]|$)
[Gg]-([^-]|$)
[Gg]#([^#]|$)
[Aa]-([^-]|$)
[Aa]#([^#]|$)
[Bb]-([^-]|$)
```

Notice that the regular expressions have been carefully defined. The first regular expression defines a pattern consisting of either an upper- or lower-case letter 'C' followed either by a character that is neither a sharp (#) nor a flat (-) nor another letter 'C', nor is followed by the end of the line ($).

Recall that parenthesis grouping (...) is part of the *extended* regular expression syntax. Therefore, we should use the **egrep** rather than the **grep** command with the above expressions. We can count the number of notes in a monophonic `**kern` input that belong to this whole-tone set:

```
egrep -c -f whole1 debussy
```

If the file `whole2` contains regular expressions for the complementary pitch set, we could similarly count the number of pitches that belong to this alternative set:

```
egrep -c -f whole2 debussy
```

Reprise

The **grep** command is usually thought of as a way to find particular patterns in a file or input stream. However, the various options for **grep** (such as -v, -l, and -L) allow **grep** to be used for other purposes. It can be used to isolate data, to count occurrences of patterns, to eliminate unwanted lines, to identify files for processing, and to avoid files that contain certain information.

We have seen how the **xargs** command can be used to carry out **AND**-searches where each work must conform to multiple criteria. We have also seen how the **-f** option for **grep** can be used to permit **OR**-searches where a work needs to conform only to one of a set of possible criteria.

Although this chapter has focussed principally on the **grep** command, the ensuing chapters will show that regular expressions are used by a wide variety of commands. In Chapter 33, many more powerful examples will be discussed in conjunction with the **find** command.