Chapter 9

Searching with Regular Expressions

A common task in computing environments is searching through some set of data for occurrences of a given pattern. When a pattern is found, various courses of action may be taken. The pattern may be copied, counted, deleted, replaced, isolated, modified, or expanded. A successful pattern match might even be used to initiate further pattern searches.

In Chapter 3 we introduced simple searching using the **grep** command. We used **grep** to search for strings of characters that match a particular pre-defined string. This chapter describes the full power of *regular expressions* for defining complex patterns of characters. Becoming skilled with regular expressions is perhaps the principal foundation for productive use of Humdrum. Regular expressions can be used to define patterns in any representation, and are widely used in many UNIX and Humdrum tools.

Regular expression syntax provides a standardized method for defining patterns of characters. Regular expressions are restricted to common text characters including the upper- and lower-case letters of the alphabet, the digits zero to nine, and other characters typically found on computer keyboards.

Regular expressions will not allow users to define every possible musical pattern of potential interest. In particular, regular expressions cannot be used directly to identify deep-structure patterns from surface-level representations. However, regular expressions are quite powerful — much more powerful than they appear to the novice user. Not all users will be equally adept at formulating an appropriate regular expression to search for a given pattern. As with the study of a musical instrument, practise is advised.

Literals

The simplest regular expressions are merely literal sequences of characters forming a character **string,** as in the pattern:

```
car
```

This pattern will match any data string containing the sequence of letters c-a-r. The letters must be contiguous, so no character (including spaces) can be interposed between any of the letters. The above pattern is present in strings such as "`carillon`" and "`ricercare`" but not in strings such as "Caruso" or "clarinet". The above pattern is called a *literal* since the matching pattern must be literally identical to the regular expression (including the correct use of upper- or lower-case).

When a pattern is found, a starting point and an ending point are identified in the input string, corresponding to the defined regular expression. The specific sequence of characters found in the input string is referred to as the *matched string* or *matched pattern.*

Wild-Card

Regular expressions that are not literal involve so-called *metacharacters.* Metacharacters are used to specify various operations, and so are not interpreted as their literal selves. The simplest regular expression metacharacter is the period (`.`). The period matches *any single character* — including spaces, tabs, and other ASCII characters. For example, the pattern:

```
c.u
```

will match any input string containing three characters, the first of which is the lower-case 'c' and the third of which is the lower-case 'u'. The pattern is present in strings such as "counterpoint" and "acoustic" but not in "cuivre" or "Crumhorn". Any character can be interposed between the 'c' and the 'u' provided there is precisely one such character.

Escape Character

A problem with metacharacters such as the period is that sometimes the user wants to use them as literals. The special meaning of metacharacters can be "turned-off" by preceding the metacharacter with the backslash character (\). The backslash is said to be an *escape* character since it is used to release the metacharacter from its special function. For example, the regular expression

```
\.
```

will match the period character. The backslash itself may be escaped by preceding it by an additional backslash (i.e. `\\`).

Repetition Operators

Another metacharacter is the plus sign (+). The plus sign means "one or more consecutive instances of the previous expression." For example,

```
fo+
```

specifies any character string beginning with a lower-case 'f' followed by one or more consecutive instances of the small letter 'o'. This pattern is present in such strings as "food" and "folly," but not in "front" or "flood." The length of the matched string is variable. In the case of "food" the matched string consists of three characters, whereas in "folly" the matched string consists of just two characters.

The plus sign in our example modifies only the preceding letter 'o' — that is, the single letter 'o' is deemed to be the *previous expression* which is affected by the +. However, the affected expression need not consist of just a single character. In regular expressions, parentheses ( ) are metacharacters that can be used to bind several characters into a single unit or sub-expression. Consider, by way of example, the following regular expression:

```
(fo)+
```

The parentheses now bind the letters 'f' and 'o' into a single two-character expression, and it is this expression that is now modified by the plus sign. The regular expression may be read as "one or more consecutive instances of the string 'fo'." This pattern is present in strings like "food" (one instance) and "fofoe" (two instances).

Of course we can mix metacharacters together. The expression:

```
(.o)+
```

will match strings such as "polo" and the first four letters of "tomorrow."

Several sub-expressions may occur within a single regular expression. For example, the following regular expression means "one or more instances of the letter 'a', followed by one or more instances of the string 'go'."

```
(a)+(go)+
```

This would match character strings in inputs such as "ago" and "agogic," but not in "largo" (intervening 'r') or "gogo" (no leading 'a'). Note that the parentheses around the letter 'a' can be omitted without changing the sense of the expression. The following expression mixes the + repetition operator with the wild-card (.):

```
c+.m+
```

This pattern is present in strings such as "accompany," "accommodate," and "cymbal." This pattern will also match strings such as "ccm" since the second 'c' can be understood to match the period metacharacter.

A second repetition operator is the asterisk (*). The asterisk means "zero or more consecutive instances of the previous expression." For example,

```
Do*r
```

specifies any character string beginning with an upper-case 'D' followed by zero or more instances of the letter 'o' followed by the letter 'r'. This pattern is present in such strings as "Dorian," "Doors" as well as "Drum," and "Drone." As in the case of the plus sign, the asterisk modifies only the preceding expression — in this case the letter 'o'. Multi-character expressions may be modified by the asterisk repetition operator by placing the expression in parentheses. Thus, the regular expression:

```
ba(na)*
```

will match strings such as "ba," "bana," "banana," "bananana," etc.

Incidentally, notice that the asterisk metacharacter can be used to replace the plus sign (+) metacharacter. For example, the regular expression `X+` is the same as `XX*`. Similarly, `(abc)+` is equivalent to `(abc)(abc)*`.

A frequent construction used in regular expressions joins the wild-card (`.`) with the asterisk repetition character (`*`). The regular expression:

```
.*
```

means "zero or more instances of any characters." (Notice the plural "characters;" this means the repetition need not be of one specific character.) This expression will match *any string*, including nothing at all (the *null string*). By itself, this expression is not very useful. However it proves invaluable in combination with other expressions. For example, the expression:

```
{.*}
```

will match any string beginning with a left curly brace and ending with a right curly brace. If we replaced the curly braces by the space character, then the resulting regular expression would match any string of characters separated by spaces — such as printed words.

A third repetition operator is the question mark (`?`), which means "zero or one instance of the preceding expression." This metacharacter is frequently useful when you want to specify the presence or absence of a single expression. For example, the pattern:

```
Ch?o
```

is present in such strings as "`Chopin`" and "`Corelli`" but not "`Chinese`" or "`cornet`."

Once again, parentheses can be used to specify more complex expressions. The pattern:

```
Ch?(o)+
```

is present in such strings as "`Chorale`," "`Couperin`," and "`Cooper`," but not in "`Chloe`" or "`Chant`."

In summary, we've identified three metacharacters pertaining to the number of occurrences of some character or string. The plus sign means "one or more," the asterisk means "zero or more," and the question mark means "zero or one." Collectively, these metacharacters are known as *repetition operators* since they indicate the number of times an expression can occur in order to match.

Context Anchors

Often it is helpful to limit the number of occurrences matched by a given pattern. You may want to match patterns in a more restricted context. One way of restricting regular expression pattern-matches is by using so-called *anchors*. There are two regular expression anchors. The caret (`^`) anchors the expression to the beginning of the line. The dollar sign (`$`) anchors the expression to the end of the line. For example,

```
^A
```

matches the upper-case letter 'A' only if it occurs at the beginning of a line. Conversely,

```
A$
```

will match the upper-case letter 'A' only if it is the last character in a line. Both anchors may be used together, hence the following regular expression matches only those lines containing just the letter 'A':

```
^A$
```

Of course anchors can be used in conjunction with the other regular expressions we have discussed. For example, the regular expression:

```
^a.*z$
```

matches any line that begins with 'a' and ends with 'z'.

OR Logical Operator

One of several possibilities may be matched by making use of the logical *OR* operator, represented by the vertical bar (|). For example, the following regular expression matches either the letter 'x' or the letter 'y' or the letter 'z':

```
x|y|z
```

Expressions may consist of multiple characters, as in the following expression which matches the string 'sharp' or 'flat' or 'natural'.

```
sharp|flat|natural
```

More complicated expressions may be created by using parentheses. For example, the regular expression:

```
(simple|compound) (duple|triple|quadruple|irregular) meter
```

will match eight different strings, including `simple triple meter` and `compound quadruple meter`.

Character Classes

In the case of single characters, a convenient way of identifying or listing a set of possibilities is to use the *character class*. For example, rather than writing the expression:

```
a|b|c|d|e|f|g
```

the expression may be simplified to:

```
[abcdefg]
```

Any character within the square brackets (a "character class") will match. Spaces, tabs, and other characters can be included within the class. When metacharacters like the period (`.`), the asterisk (*), the plus sign (+), and the dollar sign (`$`) appear within a character class, they lose their special meaning, and become simple literals. Thus the regular expression:

```
[xyz.+*$]
```

matches any one of the characters 'x,' 'y,' 'z,' the period, plus sign, asterisk, or the dollar sign.

Some other characters take on special meanings within character classes. One of these is the dash (`-`). The dash acts as a *range* operator. For example,

```
[A-Z]
```

represents the class of all upper-case letters from A to Z. Similarly,

```
[0-9]
```

represents the class of digits from zero to nine. The expression given earlier — `[abcdefg]` — can be simplified further to: `[a-g]`. Several ranges can be mixed within a single character class:

```
[a-gA-G0-9#]
```

This regular expression matches any one of the lower- or upper-case characters from A to G, or any digit, or the octothorpe (#). If the dash appears at the beginning or end of the character class, it loses its special meaning and becomes a literal dash, as in:

```
[a-gA-G0-9#-]
```

This regular expression adds the dash character to the list of possible matching characters.

Another useful metacharacter within character classes is the caret (`^`). When the caret appears at the beginning of a character-class list, it signifies a *complementary character class.* That is, only those characters *not* in the list are matched. For example,

```
[^0-9]
```

matches any character other than a digit. If the caret appears in any position other than at the beginning of the character class, it loses its special meaning and is treated as a literal. Note that if a character-class range is not specified in numerically ascending order or alphabetic order, the regular expression is considered ungrammatical and will result in an error.

Examples of Regular Expressions

The following table lists some examples of regular expressions and provides a summary description of the effect of each expression:

A	match letter 'A'
^A	match letter 'A' at the beginning of a line
A$	match letter 'A' at the end of a line
.	match any character (including space or tab)
A+	match one or more instances of letter 'A'
A?	match a single instance of 'A' or the null string
A*	match one or more instances of 'A' or the null string
.*	match any string, including the null string
A.*B	match any string starting with 'A' up to and including 'B'
A|B	match 'A' or 'B'
(A)|(B)	match 'A' or 'B'
[AB]	match 'A' or 'B'
[^AB]	match any character other than 'A' or 'B'
AB	match 'A' followed by 'B'
AB+	match 'A' followed by one or more 'B's
(AB)+	match one or more instances of 'AB', e.g. ABAB
(AB)|(BA)	match 'AB' or 'BA'
[^A]AA[^A]	match two 'A's preceded and followed by characters other than 'A's
^[^^]	match any character at the beginning of a record except the caret

Examples of regular expressions.

Examples of Regular Expressions in Humdrum

The following table provides some examples of regular expressions pertinent to Humdrum-format inputs:

^!!	match any global comment
^!!.*Beethoven	match any global comment containing 'Beethoven'
^!!.*[Rr]ecapitulation	match any global comment containing the word 'Recapitulation' or 'recapitulation'
^!($|[^!])	match only local comments
^**	match any exclusive interpretation
^*[^*]	match only tandem interpretations
^*[-+vx^]$	match spine-path indicators
^[^*!]	match only data records
^[^*!].*$	match entire data records
^(\.<*tab*>)*\.$	match records containing only null tokens (<*tab*> means a tab)
^*f#:	match key interpretation indicating F# minor

Regular expressions suitable for all Humdrum inputs.

By way of illustration, the next table shows examples of regular expressions appropriate for processing `**kern` representations.

^=	match any **kern barline or double barline
^=[^=]	match **kern single barlines but not double barlines
^[^=]	match any token other than a barline or double barline
;	match any **kern note or barline containing a pause
T	match any **kern note containing a whole-tone trill
[Tt]	match any **kern note containing a whole-tone or half-tone trill
–	match any **kern note containing at least one flat
[#]	match any **kern note containing a sharp, double-sharp, etc.
[#n–;]	match any **kern note containing an accidental, including a natural
[A-Ga-g]+	match any diatonic pitch letter-name
[0-9]+\.	match **kern dotted durations
[0-9]+\.\.[^.]	match only doubly-dotted durations
[Gg]+[^#-]	match any **kern pitch 'G' that does not have a sharp or flat
(^|[^g])gg($|[^g#-])	match only the pitch 'gg' (G5)
{.*r|r.*{	match all phrases that start with a rest
^4[^0-9.]|[^0-9]4([^0-9.]|$)	match **kern quarter durations
^(8|16)[^0-9.]|[^0-9](8|16)[^0-9.]	match eighth and sixteenth durations only
(([Ee]+-)|([Gg]+-)|([Bb]+-))($|[^-])	match any note from E-flat minor chord

Regular expressions suitable for `**kern` *data records.*

Note that the above regular expressions assume that comments and interpretations are not processed in the input. The processing of just data records can be assured by embedding each of the regular expressions given above in the expression

`(^[^*!].*`*regexp*`)|(^`*regexp*`)`

For example, the following regular expression can be used to match `**kern` trills without possibly mistaking comments or interpretations:

```
(^[^*!].*[Tt])|(^[Tt])
```

For Humdrum commands such as **humsed**, **rend**, **yank**, **xdelta**, and **ydelta**, regular expressions are applied only to data records so there is no need to use the more complex expressions. In many circumstances, we will see that it is convenient to use the Humdrum **rid** command to explicitly remove comments and interpretations prior to processing (see Chapter 13).

Basic, Extended, and Humdrum-Extended Regular Expressions

Over the years, new features have been added to regular expression syntax. Some of the early software tools that make use of regular expressions do not support the extended features provided by more recently developed tools. So-called "basic" regular expressions include the following features: the single-character wild-card (`.`), the repetition operators (`*`) and (`?`) but not (+), the context anchors (`^`) and (`$`), character classes (`[...]`), or complementary character classes (`[^...]`). Parenthesis grouping is supported in basic regular expressions, but the parentheses must be used in conjunction with the backslash to *enable* this function (i.e. \ (\ )). In Chapter 3 we introduced the **grep** command; **grep** supports only basic regular expressions.

"Extended" regular expressions include the following: the single-character wild-card (.), the repetition operators (*), (?) and (+), the context anchors (^) and ($), character classes ([...]), complementary character classes ([^...]), the logical OR (|), and parenthesis grouping. Extended regular expressions are supported by the **egrep** command; **egrep** operates in the same manner as **grep**, only the search patterns are interpreted according to extended regular expression syntax.

The Humdrum **pattern** command further extends regular expression syntax by providing multi-record repetition operators that prove very useful in musical applications. These Humdrum extensions will be discussed in Chapter 21.

Reprise

Regular expressions provide a powerful method for defining abstract patterns of alphanumeric characters. The wild card (.) matches any character. Repetition operators include "one or more" (+), "zero or more" (*), and "zero or one" (?). Context anchors define the beginning of the line (^) or the end of the line ($). Character classes ([ ]) specify a choice of possible characters. Ranges can be specified within character classes ([–]) and complementary classes may be defined ([^]). The logical OR (|) may be used in conjunction with parentheses to define more complex expressions.

There are many software tools that make use of regular expressions. The UNIX **grep** command supports standard or "basic" regular expressions. The UNIX **egrep** command supports "extended" regular expressions.

In the next chapter we will explore how regular expressions may be used in musical applications.