Section 5

Regular Expression Reference

Regular Expression Documentation

This section of the *Reference Manual* describes in detail the operation of regular expressions. Regular expressions were developed on UNIX operating systems as a generic way of defining patterns of characters. The following documentation consists of two parts: (1) a tutorial introduction to regular expressions, and (2) a comprehensive summary of the syntax.

Regular Expressions: A Tutorial Introduction

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.

Regular expression syntax provides a standardized method for defining patterns of characters. Regular expressions are restricted to common ASCII characters — including the upper- and lower-case letters of the alphabet, the digits zero to nine, and other special characters typically found on typewriter-like keyboards.† Since all Humdrum representations are based on strings of ASCII characters, regular expressions can be used to identify patterns of musical signifiers. Regular expressions can be used with any type of Humdrum representation; it does not matter what type of information is represented by the signifiers.

A number of general-purpose commands rely on regular expression syntax for specifying patterns. These include the UNIX **awk, ed, egrep, ex, expr, grep, gres, pg, sed** and **vi** commands. In addition, more than a dozen Humdrum tools rely on regular expression syntax for specifying patterns of ASCII characters. Included in these tools are the Humdrum **correl**, **fill**, **mint**, **num**, **patt**, **pattern**, **recode**, **rend**, **xdelta** and **yank** commands.

For musicologist-users interested in searching for complex music-related patterns, it is valuable to develop some facility in using regular expressions. Regular expressions will not allow users to define every possible musical pattern of potential interest. In particular, regular expressions cannot be used to identify deep-structure patterns from surface-level representations. However, regular expressions are quite powerful — typically 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:

```
foo
```

This pattern will match any data string containing the sequence of letters f-o-o. 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 "`fool`" and "`mgfooXy`" but not in strings such as "`Foo`" or "`follow`". 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).

---

† The initialism 'ASCII' stands for the American Standard Code for Information Interchange.

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

Patterns that are not literal include 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:

```
f.o
```

will match any input string containing three characters, the first of which is the lower-case '`f`' and the third of which is the lower-case '`o`'. Hence, the above pattern is present in strings such as "`flow`" and "`proof of`" but not in "`follow`" or "`found`". Any character can be interposed between the '`f`' and the '`o`' provided there is precisely just 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 normal 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.`" Notice that the length of the matched string is changeable. In the case of "`food`" the matched string consists of three characters, whereas in "`folly`" the matched string consists of just two characters. Notice also that the plus sign 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 expression, and it is this expression that is now modified by the plus sign. The above 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).

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 'bl'."

```
(a)+(bl)+
```

This would match character strings in inputs such as "able" and "kraable," but not in "dabble" (two consecutive b's) or "blbl" (no leading 'a'). Note that the parentheses are not required around the letter 'a' in the above regular 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 character.

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

```
fo*r
```

specifies any character string beginning with a lower-case 'f' followed by zero or more instances of the letter 'o' followed by the letter 'r'. This pattern is present in such strings as "ford," "fooooorm" as well as "fred," and "front." 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)*`. The plus sign (+) metacharacter is not strictly necessary, but it is frequently more convenient.

A frequent construct 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,

```
fl?o
```

will match "`flow`" and "`fodder`" but not "`fly`" or "`fllo`."

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

```
fl?(o)+
```

is present in such strings as "`flow`," "`food`," and "`flood`," but not in "`fllox`" or "`frown`."

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 know as *repetition operators* since they indicate the number of times an expression can occur in order to match.

Min-Max Character Repetition

In addition to these general repetition operators, there is a syntax to specify the precise number of occurrences for a single character, or a numerical range of possible repetitions. Three syntactical forms are provided. All three forms use the special delimiters \{ and \}. The first form specifies the exact number of repetitions:

---

† Sometimes it is difficult to remember which metacharacter has which effect. The follow mnemonic may help: question mark connotes "it's either there or it's not;" the plus sign connotes "maybe there's an addition one;" and the asterisk connotes "match the universe — including nothing."

```
x\{10\}
```

This regular expression will match precisely 10 consecutive occurrences of the lower-case letter 'x'. The second form specifies the minimum number of repetitions:

```
x\{10,\}
```

(Note the presence of the comma.) This regular expression will match an entire string of 'x's that consists of *at least* 10 consecutive occurrences of 'x'. The third form specifies a minimum and maximum number of repetitions:

```
x\{10,20\}
```

This regular expression will match a string of 'x's that consists of at least 10 occurrences, but not more than 20 occurrences of 'x'.

Priority of the Longest String Match

When matching regular expressions to some input, the operation procedes from left to right, and *longer matching strings take priority over shorter matching strings*. Consider, for example, an input record (line) consisting of 29 consecutive 'x's. Given the regular expression used above,

```
x\{10,20\}
```

the input would be parsed as follows. The first 10 'x's would satisfy the minimum length criterion and so, by themselves, would constitute a "matching string." In encountering the eleventh letter 'x', this longer string would be recognized as also satisfying the regular expression. Hence, the end-point of the matching string would be moved to the eleventh 'x' and the previous 10-character matching-string would be superceded. A similar process will continue until the twentieth consecutive letter 'x' is encountered. With the 21st 'x', the maximum length criterion prevents this 'x' from joining the preceding 20 as a matching string. Having matched this maximum length string, the regular expression parser will continue from the 21st letter 'x' to see if another matching string can be found. Since there are only nine remaining 'x's, no further matching strings exist in this input. Although it is possible to conceive of 29 'x's as two matching strings consisting of (say) 19 and 10 consecutive 'x's respectively, in this case the regular expression parser will identify only a single matching string. Once again, the longest matching string takes precedence over shorter potential matching strings.

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 string. The dollar sign ($) anchors the expression to the end of the string. For example,

```
^A
```

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

```
A$
```

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

```
^A$
```

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

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

matches any string 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 (|) metacharacter. 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 regular expression *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 in character classes, they lose their special meaning, and become simple literals. Thus the regular expression:

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

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

```
[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.

Character Class Keywords

Some character class expressions occur frequently in pattern definitions. The ten most common character class expressions can be invoked via character-class *keywords*. These are listed below:

[[:upper:]]	match any upper-case alphabetic character (same as [A-Z])
[[:lower:]]	match any lower-case alphabetic character (same as [a-z])
[[:alpha:]]	match any upper- or lower-case alphabetic character (same as [a-zA-z])
[[:digit:]]	match any single numerical digit (same as [0-9])
[[:alnum:]]	match any alphanumeric character (same as [0-9a-zA-Z])
[[:space:]]	match any empty space (blank, tab)
[[:graph:]]	match any grapheme (any character except empty space)
[[:print:]]	match any printable character (empty space included)
[[:punct:]]	match any non-alphanumeric character
[[:cntrl:]]	match any non-printable character, including control characters

Character-class keywords for regular expressions

Multiple character-class keywords can appear together, hence

```
[[:lower:][:digit:]]
```

with match any lower-case alphabetic character, or any digit.

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 string
A$	match letter 'A' at the end of a string
.	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\{5\}	match five instances of 'A'
A\{5,\}	match five or more instances of 'A'
A\{5,9\}	match between five and nine instances of 'A'
[^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($|[^g#-])	match only the pitch 'gg' (G5)
{.*r|r.*{	match all phrase beginnings that start with a rest
^4[^0-9.]|[^0-9]4([^0-9.]|$)	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, xdelta, yank,** and *ydelta,* regular expressions are applied only to data records so there is no need to use the more complex expressions. In some circumstances, the **rid** command might be used to explicitly remove comments and interpretations prior to processing.

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 ([...]), and 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. \(\)).

So-called "extended" regular expressions include the following (added features are highlighted in bold): the single-character wild-card (.), the repetition operators (*), (**?**) and (+), **min-max character repetition** (\{ \}), the context anchors (^) and ($), character classes ([...]), complementary character classes ([^...]), **character-class keywords** ([:...:]), **the logical OR** (|), and parenthesis-grouping.

Record-Repetition Operators

The Humdrum **pattern** command permits an additional regular expression feature that is especially useful in musical applications. Specifically, **pattern** permits the defining of patterns spanning more than one line or record. Record-repetition operators are specified by following the regular expression with a tab — followed by either +, *, or ?. For example, consider the following Humdrum-extension regular expression:

```
X    +
Y    *
Z    ?
```

When the metacharacters +, *, or ? appear at the end of a record, preceded by a tab character, they pertain to the number of records, rather than the number of repetitions of the expression within a record. The first record of the regular expression (X<*tab*>+) will match one or more successive lines each containing the letter 'X'. The second record of the regular expression (Y<*tab*>*) will match zero or more subsequent lines containing the letter 'Y'. The third record of the regular expression (Z<*tab*>?) will match zero or one line containing the letter 'Z'. Hence, the above multi-record regular expression would match an input such as the following: three successive lines containing the letter 'X', followed by eight successive lines containing the letter 'Y', followed by a single line containing the letter 'Z'. Similarly, the above regular expression would match an input containing one line containing the letter 'X'.

Record-repetition operators can be used in conjunction with all of the other regular expression features. For example, the following regular expression matches one or more successive **kern data records containing the pitch 'G' (naturals only) followed optionally by a single 'G#' followed by one or more records containing one or more pitches from an A major triad — the last of which must end a phrase:

```
[Gg]+[^#-]     +
[Gg]+#[^#]     ?
([Aa]+|([Cc]+#)|[Ee]+)[^#-]          *
(}.*([Aa]+|([Cc]+#)|[Ee]+)[^#-]))|(([Aa]+|([Cc]+#)|[Ee]+)[^#-].*})
```

NAME

regexp — regular expression pattern-match syntax

DESCRIPTION

"Regular expression syntax" provides a standardized way of defining pattern of characters. Regular expressions are limited to common ASCII characters — such as the letters of the alphabet, numbers, and other special characters typically found on typewriter-like keyboards.

Three variants of regular expression syntax can be distinguished: (1) basic regular expressions, (2) extended regular expressions, and (3) Humdrum-extended regular expressions. The differences are outlined later in this documentation.

SYNTAX

A regular expression is any combination of sub-expressions consisting of literals, wild cards, repetition operators, min-max character repetition, context anchors, character classes, complementary character classes, the logical OR, and parenthesis-grouping.

Literals. A literal is any string not containing unescaped metacharacters. Metacharacter may be treated as literal characters by preceding the metacharacter by the escape character — the backslash (\). (The backslash itself may be escaped by preceding it by an additional backslash.) Literals are matched only if a string of characters is found that is identical to the literal.

Wild Card (.) The period character (.) is a wild-card that matches any single character.

Repetition Operators (+) (?) (*). An *expression* followed by the plus sign (+) matches any string of one or more occurrences of *expression*. An *expression* followed by the question mark (?) matches zero or one occurrence of *expression*. An *expression* followed by the asterisk (*) matches zero or more occurrences of *expression*.

Min-Max Character Repetition (\{ \}). Precise numbers of occurrences for a single character, or minimum and maximum numbers of occurrences can be specified using the special delimiters \{ and \}. Three syntactical forms exist; the regular expression `x\{10\}` will match precisely 10 occurrences of the lower-case letter 'x'. The regular expression `x\{10,\}` will match an entire string of 'x's that consists of *at least* 10 occurrences of 'x'. The regular expression `x\{10,20\}` will match a string of 'x's that consists of at least 10 occurrences, but not more than 20 occurrences of 'x'.

**Context Anchors (^) ($).** An *expression* preceded by the caret anchor (^) matches only those occurrences of *expression* starting at the beginning of a string. An *expression* followed by the dollar sign anchor ($) matches only those occurrences of *expression* ending at the end

of a line.

Character Classes ([...]). Any character given in a string bounded by left '[' and right ']' braces will be matched. Character ranges can be indicated via the dash sign (-), hence the character class [A-Z] matches any upper-case letter, and [5-8] matches any of the digits 5, 6, 7, or 8. If the dash sign is placed at the beginning or the end of the character class, it loses its special meaning; hence [+-] matches the plus or minus sign. The metacharacters `*`, `+`, `?`, `$`, `(`, and `)` lose their special meanings within character classes.

Complementary Character Classes ([^...]). If the first character in a character class is the caret (^), the character class is negated. Only those characters *not* in the character class will produce a match.

Logical OR (|). Two or more *expressions* separated by | will cause matches for any of the component expressions. Hence the regular expression `abc|lmn|xyz` will match either '`abc`,' '`lmn`,' or '`xyz`.'

Parenthesis Grouping ( ). Expressions may be logically grouped using parentheses. Hence, the expression `(abc)+` means one or more occurrences of the string '`abc`'. Multiple levels of grouping are possible such as `((abc|DEF)+(xyz))+` — which matches strings such as "`abcabcxyz`" and "`abcxyzDEFDEFabcxyz`."

EXAMPLES

A		match letter 'A'
^A		match letter 'A' at the beginning of a string
A$		match letter 'A' at the end of a string
.		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\{5\}		match five instances of 'A'
A\{5,\}		match five or more instances of 'A'
A\{5,9\}		match between five and nine instances of 'A'
[^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
A	+	match one or more lines containing the letter 'A'
A	?	match zero or one line containing the letter 'A'
AA+	*	match zero or more lines containing at least two consecutive 'A's

Examples of regular expressions.

VARIANTS

Three variants of regular expression syntax exist: (1) basic, (2) extend, and (3) Humdrum-extended. "Basic" regular expressions include the following features: the single-character wild-card (`.`), the repetition operator (`*`) but not (`?`) or (`+`), the context anchors (`^`) and (`$`), character classes (`[...]`), and 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. `\ ( \ )`).

"Extended" regular expressions include the following (added features are highlighted in bold): the single-character wild-card (`.`), the repetition operators (`*`), (`?`) and (`+`), **min-max character repetition** (`\{ \}`), the context anchors (`^`) and (`$`), character classes (`[...]`), complementary character classes (`[^...]`), **character-class keywords ([[:...:]]), the logical OR** (`|`), and parenthesis-grouping.

The Humdrum-extended syntax allows the defining of patterns spanning more than one line or record. When the metacharacters `+`, `*`, or `?` appear at the end of a record, preceded by a tab character, they are **record-repetition operators** and pertain to the number of records, rather than the number of repetitions of the expression within a record. For example, the

letter 'A' followed by a tab, followed by + means one or more records containing the letter 'A'. (This syntax is only available with the ~~H~~humdrum **pattern** command.)

COMMANDS

A number of Humdrum commands make use of regular expression syntax, including **correl**, **fields**, **fill**, **hint**, **mint**, **num**, **patt**, **pattern**, **recode**, **regexp**, **rend**, **scramble**, **xdelta**, **yank**, and **ydelta**.

In addition, the following UNIX commands make use of regular expression syntax:

awk	pattern-action language
ed	line-oriented text editor
ex	text editor
expr	shell expression evaluator
grep	pattern-match command
gres	pattern substitution command
pg	interactive text display
sed	stream editor
vi	interactive full-screen text editor

UNIX commands employing regular expressions.

SEE ALSO

awk (UNIX), **ed** (UNIX), **egrep** (UNIX), **ex** (UNIX), **expr** (UNIX), **extract** (4), **fields** (4), **grep** (UNIX), **gres** (UNIX), **humsed** (4), **patt** (4), **pattern** (4), **pg** (UNIX), **recode** (4), **rend** (4), **sed** (UNIX), **vi** (UNIX), **xdelta** (4), **yank** (4), **ydelta** (4)