Introduction to Regular Expressions used in R

Description

This short note, based on help document for regexp() the regular expression patterns supported by grep() and related functions grepl(), regexpr(), gregexpr(), sub() , gsub() and strsplit().

Details

A **regular expression** is a pattern that describes a set of strings. Two types of regular expressions are used in **R**, *extended* regular expressions (the default) and *Perl-like* regular expressions used by perl = TRUE. There is a also fixed = TRUE which can be considered to use a *literal* regular expression.

Other functions which use regular expressions (often via the use of grep()) include apropos(), browseEnv(), help.search(), list.files() and ls(). These will all use *extended* regular expressions.

Patterns are described here as they would be printed by cat: (*do remember that backslashes need to be doubled when entering **R** character strings*, e.g. from the keyboard).

Long regular expressions may or may not be accepted: the POSIX standard only requires up to 256 *bytes*.

Extended Regular Expressions

This section covers the regular expressions allowed in the default mode of grep(), regexpr(), gregexpr(), sub(), gsub() and strsplit().

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. The whole expression matches zero or more characters (read ‘character’ as ‘byte’ if useBytes = TRUE).

The fundamental building blocks are the regular expressions that match a single character.

Most characters, including all letters and digits, are regular expressions that match themselves.

Any ***metacharacter*** with special meaning may be quoted by preceding it with a backslash. The *metacharacters* in extended regular expressions are

\ | ( ) [{ ^ $ * + ?

But note that whether these have a special meaning depends on the context.

Escaping *non-metacharacters* with a backslash is implementation-dependent. The current implementation interprets **\a** as BEL, **\e** as ESC, **\f** as FF, **\n** as LF, **\r** as CR and **\t** as TAB. (Note that these will be interpreted by **R**'s parser in literal character strings.)

A ***character class*** is a list of characters enclosed between **[** and **]** which matches any single character in that list; unless the first character of the list is the caret **^**, when it matches any character *not* in the list. For example, the regular expression **[0123456789]** matches any single digit, and **[^abc]** matches anything except the characters a, b or c.

A *range of characters* may be specified by giving the first and last characters, separated by a

hyphen. (Because their interpretation is locale- and implementation-dependent, character ranges are best avoided.)

The only portable way to specify all ASCII letters is to list them all as the character class
[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]
(The current implementation uses numerical order of the encoding.)

Certain named classes of characters are predefined. Their interpretation depends on the *locale*; the interpretation below is that of the POSIX locale.

[:alnum:] Alphanumeric characters: **[:alpha:]** and **[:digit:]**

[:alpha:] Alphabetic characters: [:lower:] and [:upper:].

[:blank:] Blank characters: space and tab, and possibly other locale-dependent characters such as non-breaking space.

[:cntrl:] Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In another character set, these are the equivalent characters, if any.

[:digit:] Digits: **0 1 2 3 4 5 6 7 8 9**

[:graph:] Graphical characters: **[:alnum:]** and **[:punct:]**

[:lower:] Lower-case letters in the current locale.

[:print:] Printable characters: **[:alnum:]**, **[:punct:]** and space.

[:punct:] Punctuation characters: **! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~**

[:space:] Space characters: tab, newline, vertical tab, form feed, carriage return, space and possibly other locale-dependent characters.

[:upper:] Upper-case letters in the current locale.

[:xdigit:] Hexadecimal digits: **0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f**

For example, **[[:alnum:]]** means **[0-9A-Za-z]**, except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside a character class.

To include a literal **]**, place it first in the list. Similarly, to include a literal **^**, place it anywhere but first. Finally, to include a literal **-**, place it first or last (or, for perl = TRUE only, precede it by a backslash). (Only **^ - \ ]** are special inside character classes.)

The period **.** matches any single character.

The symbol **\w** matches a 'word' character (a synonym for [[:alnum:]_], an extension) and **\W** is its negation (**[^[:alnum:] _]**).

Symbols **\d**, **\s**, **\D** and **\S** denote the digit and space classes and their negations (these are all extensions).

The caret **^** and the dollar sign **$** are metacharacters that respectively match the empty string at the beginning and end of a line.

The symbols **\<** and **\>** match the empty string at the beginning and end of a word.

The symbol **\b** matches the empty string at either edge of a word, and **\B** matches the empty string provided it is not at an edge of a word. (The interpretation of 'word' depends on the locale and implementation: these are all extensions.)

A regular expression may be followed by one of several ***repetition quantifiers***:

? The preceding item is optional and will be matched at most once.

***** The preceding item will be matched zero or more times.

+ The preceding item will be matched one or more times.

{n} The preceding item is matched exactly n times.

{n,} The preceding item is matched n or more times.

{n,m} The preceding item is matched at least n times, but not more than m times.

By default repetition is greedy, so the maximal possible number of repeats is used. This can be changed to 'minimal' by appending **?** to the quantifier. (There are further quantifiers that allow approximate matching: see the TRE documentation.)

Regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating the substrings that match the concatenated subexpressions.

Two regular expressions may be joined by the ***infix operator*** **|** ; the resulting regular expression matches any string matching either subexpression. For example, abba**|**cde matches either the string abba or the string cde. Note that alternation does not work inside character classes, where **|** has its literal meaning.

Repetition *takes precedence over* ***concatenation****, which in turn takes precedence over* ***alternation****. A whole subexpression may be enclosed in parentheses to override these precedence rules.*

The ***backreference*** **\N**, where N = 1 ... 9, matches the substring previously matched by the Nth parenthesized subexpression of the regular expression. (This is an extension for extended regular expressions: POSIX defines them only for basic ones.)

See Also

grep, apropos, browseEnv, glob2rx, help.search, list.files, ls and strsplit.

The TRE documentation at http://laurikari.net/tre/documentation/regex-syntax/).

The POSIX 1003.2 standard at
http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html

The pcrepattern man page (found as part of http://www.pcre.org/original/pcre.txt), and details of Perl's own implementation at http://perldoc.perl.org/perlre.html.

Glossary

Assertions

```
assertion ::= "^"
            |   "$"
            |   "\" assertion-character
```

The expressions **^** and **$** are called “left anchor” and “right anchor”, respectively. The left anchor matches the empty string at the beginning of the string. The right anchor matches the empty string at the end of the string. The behavior of both anchors can be varied by specifying certain execution and compilation flags.

An assertion-character can be any of the following:

- **\<** Beginning of word
- **\>** End of word
- **\b** Word boundary
- **\B** Non-word boundary
- **\d** Digit character (equivalent to **[[:digit:]]**)
- **\D** Non-digit character (equivalent to **[^[:digit:]]**)
- **\s** Space character (equivalent to **[[:space:]]**)
- **\S** Non-space character (equivalent to **[^[:space:]]**)
- **\w** Word character (equivalent to **[[:alnum:]_]**)
- **\W** Non-word character (equivalent to **[^[:alnum:]_]**)

Literals

A literal is either an ordinary character (a character that has no other significance in the context), an 8 bit hexadecimal encoded character (e.g. \x1B), a wide hexadecimal encoded character (e.g. \x{263a}), or an escaped character. An escaped character is a \ followed by any character, and matches that character. Escaping can be used to match characters which have a special meaning in regexp syntax. A \ cannot be the last character of an ERE. Escaping also allows you to include a few non-printable characters in the regular expression. These special escape sequences include:

- **\a** Bell character (ASCII code 7)
- **\e** Escape character (ASCII code 27)
- **\f** Form-feed character (ASCII code 12)
- **\n** New-line/line-feed character (ASCII code 10)
- **\r** Carriage return character (ASCII code 13)
- **\t** Horizontal tab character (ASCII code 9)

An ordinary character is just a single character with no other significance, and matches that character. A **{** followed by something else than a digit is considered an ordinary character.

Bracket expressions

```
bracket-expression ::= "[" item+ "]"
                   |   "[^" item+ "]"
```

A bracket expression specifies a set of characters by enclosing a nonempty list of items in brackets. Normally anything matching any item in the list is matched. If the list begins with ^ the meaning is negated; any character matching no item in the list is matched.

Approximate matching settings

```
approx-settings ::= "{" count-limits* ","? cost-equation? "}"

count-limits ::= "+" number?
               |   "-" number?
               |   "#" number?
               |   "~" number?

cost-equation ::= ( cost-term "+"? " "? )+ "<" number

cost-term ::= number "i"
            |   number "d"
            |   number "s"
```

The approximate matching settings for a subpattern can be changed by appending *approx-settings* to the subpattern. Limits for the number of errors can be set and an expression for specifying and limiting the costs can be given.

The *count-limits* can be used to set limits for the number of insertions (**+**), deletions (**-**), substitutions (**#**), and total number of errors (**~**). If the *number* part is omitted, the specified error count will be unlimited.

The *cost-equation* can be thought of as a mathematical equation, where **i**, **d**, and **s** stand for the number of ***insertions***, ***deletions***, and ***substitutions***, respectively. The equation can have a multiplier for each of **i**, **d**, and **s**. The multiplier is the cost of the error, and the number after **<** is the maximum allowed cost of a match. Spaces and pluses can be inserted to make the equation readable. In fact, when specifying only a cost equation, adding a space after the opening **{** is required.

Examples:

{~} Sets the maximum number of errors to unlimited.

{~3} Sets the maximum number of errors to three.

{+2~5} Sets the maximum number of errors to five, and the maximum number of insertions to two.

{<3} Sets the maximum cost to three.

{ 2i + 1d + 2s < 5 } Sets the cost of an insertion to two, a deletion to one, a substitution to two, and the maximum cost to five.

Back references

back-reference ::= **"\" ["1"-"9"]**

A back reference is a backslash followed by a single non-zero decimal digit *d*. It matches the same sequence of characters matched by the *d*th parenthesized subexpression.

Options

options ::= **["i" "n" "r" "U"]*** **("-" ["i" "n" "r" "U"]*)?**

Options allow compile time options to be turned on/off for particular parts of the regular expression. The options equate to several compile time options specified to the regcomp API function. If the option is specified in the first section, it is turned on. If it is specified in the second section (after the -), it is turned off.

i Case insensitive.

n Forces special handling of the new line character.

r Causes the regex to be matched in a right associative manner rather than the normal left associative manner.

U Forces repetition operators to be non-greedy unless a **?** is appended.