Regular Expression Overview

Regular expression matching allows you to test whether a string fits into a specific syntactic shape. You can also search a string for a substring that fits a pattern.

A regular expression describes a set of strings. The simplest case is one that describes a particular string; for example, the string 'foo' when regarded as a regular expression matches 'foo' and nothing else. Nontrivial regular expressions use certain special constructs so that they can match more than one string. For example, the regular expression 'foo\|bar' matches either the string 'foo' or the string 'bar'; the regular expression 'c[ad]*r' matches any of the strings 'cr', 'car', 'cdr', 'caar', 'cadddar' and all other such strings with any number of 'a''s and 'd''s.

The first step in matching a regular expression is to compile it. You must supply the pattern string and also a pattern buffer to hold the compiled result. That result contains the pattern in an internal format that is easier to use in matching.

Having compiled a pattern, you can match it against strings. You can match the compiled pattern any number of times against different strings.

Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are "ordinary". An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '\$', '^', '.', '*', '+', '?', '[', ']' and '\'. Any other character appearing in a regular expression is ordinary, unless a '\' precedes it.

For example, 'f' is not a special character, so it is ordinary, and therefore 'f' is a regular expression that matches the string 'f' and no other string. (It does *not* match the string 'ff'.) Likewise, 'o' is a regular expression that matches only 'o'.

Any two regular expressions A and B can be concatenated. The result is a regular expression which matches a string if A matches some amount of the beginning of that string and B matches the rest of the string.

As a simple example, we can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. Still trivial.

Note: for Unix compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '*foo' treats '*' as ordinary since there is no preceding expression on which the '*' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

The following are the characters and character sequences which have special meaning within regular expressions. Any character not mentioned here is not special; it stands for exactly itself for the purposes of searching and matching.

- `.` is a special character that matches anything except a newline. Using concatenation, we can make regular expressions like `a.b` which matches any three-character string which begins with `a` and ends with `b`.

- `*` is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In `fo*`, the `*` applies to the `o`, so `fo*` matches `f` followed by any number of `o`'s.

  The case of zero `o`'s is allowed: `fo*` does match `f`.

  `*` always applies to the *smallest* possible preceding expression. Thus, `fo*` has a repeating `o`, not a repeating `fo`.

  The matcher processes a `*` construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the `*`'d construct in case that makes it possible to match the rest of the pattern. For example, matching `c[ad]*ar` against the string `caddaar`, the `[ad]*` first matches `addaa`, but this does not allow the next `a` in the pattern to match. So the last of the matches of `[ad]` is undone and the following `a` is tried again. Now it succeeds.

- `+` is like `*` except that at least one match for the preceding pattern is required for `+`. Thus, `c[ad]+r` does not match `cr` but does match anything else that `c[ad]*r` would match.

- `?` is like `*` except that it allows either zero or one match for the preceding pattern. Thus, `c[ad]?r` matches `cr` or `car` or `cdr`, and nothing else.

- `[` begins a "character set", which is terminated by a `]`. In the simplest case, the characters between the two form the set. Thus, `[ad]` matches either `a` or `d`, and `[ad]*` matches any string of `a`'s and `d`'s (including the empty string), from which it follows that `c[ad]*r` matches `car`, etc.

  Character ranges can also be included in a character set, by writing two characters with a `-` between them. Thus, `[a-z]` matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in `[a-z$%.]`, which matches any lower case letter or `$`, `%` or period.

  Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: `]`, `-` and ^.

  To include a `]` in a character set, you must make it the first character. For example, `[]a]` matches `]` or `a`. To include a `-`, you must use it in a context where it cannot possibly indicate a range: that is, as the first character, or immediately after a range.

- [^ begins a "complement character set", which matches any character except the ones specified. Thus, [^a-z0-9A-Z] matches all characters *except* letters and digits.

- ^ is not special in a character set unless it is the first character. The character following the ^ is treated as if it were first (it may be a `-` or a `]`).

  ^ is a special character that matches the empty string -- but only if at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, ^foo matches a `foo` which occurs at the beginning of a line.

- `$` is similar to ^ but matches only at the end of a line. Thus, `xx*$` matches a string of one or more `x`'s at the end of a line.

Regular Expressions http://www.uky.edu/AS/Classics/regex.html

- \ has two functions: it quotes the above special characters (including \), and it introduces additional special constructs.

  Because \ quotes special characters, `\$` is a regular expression which matches only $, and `\\`

  `[` is a regular expression which matches only [, and so on.

  For the most part, \ followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by \, are special constructs. Such characters are always ordinary when encountered on their own.

  No new special characters will ever be defined. All extensions to the regular expression syntax are made by defining new two-character constructs that begin with \.

- `\|` specifies an alternative. Two regular expressions A and B with `\|` in between form an expression that matches anything that either A or B will match.

  Thus, `foo\|bar` matches either `foo` or `bar` but no other string.

  `\|` applies to the largest possible surrounding expressions. Only a surrounding `\( ... \)` grouping can limit the grouping power of `\|`.

  Full backtracking capability exists when multiple `\|`'s are used.

- `\( ... \)` is a grouping construct that serves three purposes:
  1. To enclose a set of `\|` alternatives for other operations. Thus, `\(foo\|bar\)x` matches either `foox` or `barx`.
  2. To enclose a complicated expression for the postfix * to operate on. Thus, `ba\(na\)*` matches `bananana`, etc., with any (zero or more) number of `na`'s.
  3. To mark a matched substring for future reference.

  This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same `\( ... \)` construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

- `\DIGIT` After the end of a `\( ... \)` construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use \ followed by DIGIT to mean "match the same text matched the DIGIT'th time by the `\( ... \)` construct." The `\( ... \)` constructs are numbered in order of commencement in the regexp.

  The strings matching the first nine `\( ... \)` constructs appearing in a regular expression are assigned numbers 1 through 9 in order of their beginnings. `\1` through `\9` may be used to refer to the text matched by the corresponding `\( ... \)` construct.

  For example, `\(.*\)\1` matches any string that is composed of two identical halves. The `\(.*\)` matches the first half, which may be anything, but the `\1` that follows must match the same exact text.

- `\b` matches the empty string, but only if it is at the beginning or end of a word. Thus, `\bfoo\b` matches any occurrence of `foo` as a separate word. `\bball\(s\|\)\b` matches `ball` or `balls` as a separate word.

- `\B` matches the empty string, provided it is *not* at the beginning or end of a word.

- `\<` matches the empty string, but only if it is at the beginning of a word.

- `\>` matches the empty string, but only if it is at the end of a word.

- `\w` matches any word-constituent character.

- `\W` matches any character that is not a word-constituent.

Regular Expressions http://www.uky.edu/AS/Classics/regex.html