SWI-Prolog Regular Expression library

Jan Wielemaker
VU University Amsterdam
The Netherlands
E-mail: J.Wielemaker@vu.nl

June 8, 2017

Abstract

The library pcre provides access to Perl Compatible Regular Expressions.

Contents

1 Motivation

The core facility for string matching in Prolog is provided by DCG (*Definite Clause Grammars*). Using DCGs is typically more verbose but gives reuse, modularity, readability and mixing with arbitrary Prolog code in return. Supporting regular expressions has some advantages: (1) in simple cases the terse specification of a regular expression is more comfortable, (2) many programmers are familar with them and (3) regular expressions are part of domain specific languages one may wish to implement in Prolog, e.g., SPARQL.

There are roughly three options for adding regular expressions to Prolog. One is to simply interpret them in Prolog. Given Prolog's unification and backtracking facilities this is remarkable simple and performs quite reasonable. Still, the implementing all facilities of modern regular expression engines requires significant effort. Alternatively, we can *compile* them into DCGs. This brings terse expressions to DCGs while staying in the same framework. The disadvantage is that regular expressions become programs that are hard to reclaim, making this approach less attractive for applications that potentially execute many different regular expressions. The final option is to wrap an existing regular expression engine. This provides access to a robust implementation for which we only have to document the Prolog binding. That is the option taken by library `pcre`.

2 library(pcre): Perl compatible regular expression matching for SWI-Prolog

See also 'man pcre' for details.

This module provides an interface to the PCRE (Perl Compatible Regular Expression) library. This Prolog interface provides an almost comprehensive wrapper around PCRE.

Regular expressions are created from a pattern and options and represented as a SWI-Prolog *blob*. This implies they are subject to (atom) garbage collection. Compiled regular expressions can safely be used in multiple threads. Most predicates accept both an explicitly compiled regular expression, a pattern or a term Pattern/Flags. In the latter two cases a regular expression *blob* is created and stored in a cache. The cache can be cleared using `re_flush/0`.

re_match(+*Regex*, +*String*) *[semidet]*
re_match(+*Regex*, +*String*, +*Options*) *[semidet]*

Succeeds if *String* matches *Regex*. For example:

```
?- re_match("^needle"/i, "Needle in a haystack").
true.
```

Options:

anchored(*Bool*)
If `true`, match only at the first position

bol(*Bool*)
Subject string is the beginning of a line (default `false`)

bsr(*Mode*)
: If `anycrlf`, \R only matches CR, LF or CRLF. If `unicode`, \R matches all Unicode line endings. Subject string is the end of a line (default `false`)

empty(*Bool*)
: An empty string is a valid match (default `true`)

empty_atstart(*Bool*)
: An empty string at the start of the subject is a valid match (default `true`)

eol(*Bool*)
: Subject string is the end of a line (default `false`)

newline(*Mode*)
: If `any`, recognize any Unicode newline sequence, if `anycrlf`, recognize CR, LF, and CRLF as newline sequences, if `cr`, recognize CR, if `lf`, recognize LF and finally if `crlf` recognize CRLF as newline.

start(+*From*)
: Start at the given character index

Arguments

Regex	is the output of `re_compile/3`, a pattern or a term Pattern/Flags, where Pattern is an atom or string. The defined flags and there related option for `re_compile/3` are below.

- **x**: `extended(true)`
- **i**: `caseless(true)`
- **m**: `multiline(true)`
- **s**: `dotall(true)`
- **a**: `capture_type(atom)`
- **r**: `capture_type(range)`
- **t**: `capture_type(term)`

re_matchsub(+*Regex*, +*String*, -*Sub:dict*, +*Options*) *[semidet]*

Match *String* against *Regex*. On success, *Sub* is a dict containing integer keys for the numbered capture group and atom keys for the named capture groups. The associated value is determined by the `capture_type(Type)` option passed to `re_compile/3`, may be specified using flags if *Regex* is of the form Pattern/Flags and may be specified at the level of individual captures using a naming convention for the caption name. See `re_compile/3` for details.

The example below exploits the typed groups to parse a date specification:

```
?- re_matchsub("(?<date> (?<year_I>(?:\\d\\d)?\\d\\d) -
                 (?<month_I>\\d\\d) - (?<day_I>\\d\\d) )"/e,
               "2017-04-20", Sub, []).
Sub = re_match{0:"2017-04-20", date:"2017-04-20",
               day:20, month:4, year:2017}.
```

re_foldl(*:Goal, +Regex, +String, ?V0, ?V, +Options*) *[semidet]*

Fold all matches of *Regex* on *String*. Each match is represented by a dict as specified for `re_matchsub/4`. *V0* and *V* are related using a sequence of invocations of *Goal* as illustrated below.

```
call(Goal, Dict1, V0, V1),
call(Goal, Dict2, V1, V2),
...
call(Goal, Dictn, Vn, V).
```

This predicate is used to implement `re_split/4` and `re_replace/4`. For example, we can count all matches of a *Regex* on *String* using this code:

```
re_match_count(Regex, String, Count) :-
    re_foldl(increment, Regex, String, 0, Count, []).

increment(_Match, V0, V1) :-
    V1 is V0+1.
```

After which we can query

```
?- re_match_count("a", "aap", X).
X = 2.
```

re_split(*+Pattern, +String, -Split:list*) *[det]*
re_split(*+Pattern, +String, -Split:list, +Options*) *[det]*

Split String using the regular expression *Pattern*. *Split* is a list of strings holding alternating matches of *Pattern* and skipped parts of the *String*, starting with a skipped part. The *Split* lists ends with a string of the content of *String* after the last match. If *Pattern* does not appear in *String*, *Split* is a list holding a copy of *String*. This implies the number of elements in *Split* is *always* odd. For example:

```
?- re_split("a+", "abaac", Split, []).
Split = ["","a","b","aa","c"].
?- re_split(":\\s*"/n, "Age: 33", Split, []).
Split = ['Age', ': ', 33].
```

Arguments

Pattern	is the pattern text, optionally follows by /Flags. Similar to `re_matchsub/4`, the final output type can be controlled by a flag `a` (atom), `s` (string, default) or `n` (number if possible, atom otherwise).

re_replace(*+Pattern, +With, +String, -NewString*)

Replace matches of the regular expression *Pattern* in *String* with *With*. *With* may reference captured substrings using \N or $Name. Both N and Name may be written as {N} and {Name} to avoid ambiguities.

Arguments

Pattern	is the pattern text, optionally follows by /Flags. Flags may include `g`, replacing all occurences of *Pattern*. In addition, similar to `re_matchsub/4`, the final output type can be controlled by a flag `a` (atom) or `s` (string, default).

re_compile(*+Pattern, -Regex, +Options*) *[det]*

Compiles *Pattern* to a *Regex blob* of type `regex` (see `blob/2`). Defined *Options* are defined below. Please consult the PCRE documentation for details.

anchored(*Bool*)
Force pattern anchoring

bsr(*Mode*)
If `anycrlf`, \R only matches CR, LF or CRLF. If `unicode`, \R matches all Unicode line endings.

caseless(*Bool*)
If `true`, do caseless matching.

dollar_endonly(*Bool*)
If `true`, $ not to match newline at end

dotall(*Bool*)
If `true`, . matches anything including NL

dupnames(*Bool*)
If `true`, allow duplicate names for subpatterns

extended(*Bool*)
If `true`, ignore white space and # comments

extra(*Bool*)
If `true`, PCRE extra features (not much use currently)

firstline(*Bool*)
If `true`, force matching to be before newline

compat(*With*)
If `javascript`, JavaScript compatibility

multiline(*Bool*)
If `true`, ^ and $ match newlines within data

newline(*Mode*)
If `any`, recognize any Unicode newline sequence, if `anycrlf` (default), recognize CR, LF, and CRLF as newline sequences, if `cr`, recognize CR, if `lf`, recognize LF and finally if `crlf` recognize CRLF as newline.

ucp(*Bool*)
If `true`, use Unicode properties for \d, \w, etc.

ungreedy(*Bool*)
If `true`, invert greediness of quantifiers

In addition to the options above that directly map to pcre flags the following options are processed:

optimize(*Bool*)
: If `true`, *study* the regular expression.

capture_type(+*Type*)
: How to return the matched part of the input and possibly captured groups in there. Possible values are:

  string
  : Return the captured string as a string (default).

  atom
  : Return the captured string as an atom.

  range
  : Return the captured string as a pair `Start-Length`. Note the we use Start-Length‘ rather than the more conventional `Start-End` to allow for immediate use with `sub_atom/5` and `sub_string/5`.

  term
  : Parse the captured string as a Prolog term. This is notably practical if you capture a number.

The `capture_type` specifies the default for this pattern. The interface supports a different type for each *named* group using the syntax `(?<name_T>...)`, where `T` is one of `S` (string), `A` (atom), `I` (integer), `F` (float), `N` (number), `T` (term) and `R` (range). In the current implementation `I`, `F` and `N` are synonyms for `T`. Future versions may act different if the parsed value is not of the requested numeric type.

re_flush
: Clean pattern and replacement caches.

  To be done Flush automatically if the cache becomes too large.

re_config(+*Term*)
: Extract configuration information from the pcre library. *Term* is of the form Name(Value). Name is derived from the `PCRE_CONFIG_*` constant after removing =PCRE_CONFIG_= and mapping the name to lower case, e.g. `utf8`, `unicode_properties`, etc. Value is either a Prolog boolean, integer or atom.

  Finally, the functionality of `pcre_version()` is available using the configuration name `version`.

  See also ‘man pcreapi‘ for details

Index