module Pesco.Regex

— Regular expression matching “better than Perl” —

Sven Moritz Hallberg `<pesco@gmx.de>`

December 6th, 2004

Abstract

This document is a literate Haskell module. It wraps Text.Regex. It exposes functions for compiling, matching, and substitution. The functions are overloaded on the type of thing to match against, so strings or compiled regexes can be passed interchangeably wherever a regular expression is expected. The substitution operator is a polyvariadic function taking any combination of replacement strings and submatch references (Ints) as arguments, thus avoiding errors from parsing or constructing a replacement string with escape characters.

```
 {-# OPTIONS -fglasgow-exts #-}
 {-Documentation for this module can be found in the doc directory in the MissingH distribution∘ -}
module MissingH.Regex.Pesco
       (    Regex (match)   -- type class
       ,    Match (..)      -- data type
       ,    Subst           -- type class
       ,    (≃),   (⋍)
       ,    ($~),  (~$)
       ,    (//~), (~//)
       ,    (/~),  (~/)
       ,    CRegex          -- data type
       ,    Rexopt (..)     -- data type
       ,    cregex
       ,    subst
       ,    subst1
       ,    test            -- to be removed
       )
   where
import qualified Text.Regex as TR
import Data.Maybe (isJust)
import Data.List (unfoldr)
```

Motivation

When asked the inevitable[1] by a Perl programmer, what do we answer?

> Of course it does, it uses the POSIX regex library, just import Text.Regex, and have a look at *mkRegex* and *matchRegex*...

which to the Perl programmer must sound like "Basically, it works as in C". Therefore I'd like to answer instead

> Basically, it works just as in Perl.

followed by appropriate mumbling about strong typing and syntax aesthetics.

Well, of course Haskell neither can nor should absolutely resemble Perl. I've tried to catch the essence that makes the use of regular expressions so easy in Perl while still doing so in what a prototypical Haskell programmer could consider "the right way".

Overview

Motivated by the above, I export operators for the common regex operations:

$s \simeq r$ tests whether string s matches the regular expression r.

$$(\simeq) :: (\mathrm{Regex}\ \rho) \Rightarrow \mathrm{String} \to \rho \to \mathrm{Bool}$$

Notice the type class Regex. It alleviates the need to explicitly "compile" or "make" regexes. You can pass compiled expressions or plain strings anywhere a Regex is expected.

$s\ \$\tilde{}\ r$ applies regex r to the string s, yielding the list of all matches.

$$(\$\tilde{}) :: (\mathrm{Regex}\ \rho) \Rightarrow \mathrm{String} \to \rho \to [\mathrm{Match}]$$

The Match data type will be defined shortly. It's a record telling which substring of s matched, as well as any subexpression matches.

$(s\ //\tilde{}\ r)\ p\texttt{...}$ replaces any match of r in s with pattern $p\ldots$.

$$(//\tilde{}) :: (\mathrm{Regex}\ \rho, \mathrm{Subst}\ \pi) \Rightarrow \mathrm{String} \to \rho \to \pi$$

Notice the type class Subst. This operator takes a variable number of arguments of possibly different types. The mechanism will become clear when class Subst is defined. The effect, anyway, is that $p\ldots$ in the above can be an arbitrary sequence of String or Int arguments. The Ints represent submatch references, so for example,

$$\mathit{test} = (\texttt{"Hello, World!"}\ //\tilde{}\ \texttt{"W(o)rld"})\ \texttt{"Hell"}\ (1 :: \mathrm{Int}) :: \mathrm{String}$$

yields `"Hello, Hello!"`.

$(s\ /\tilde{}\ r)\ p\texttt{...}$ is like $//\tilde{}$ but replaces only the first match.

$$(/\tilde{}) :: (\mathrm{Regex}\ \rho, \mathrm{Subst}\ \pi) \Rightarrow \mathrm{String} \to \rho \to \pi$$

[1] "Does it support regexes?"

In addition to the above, each operator has a “flipped” sibling, the rule being that “the pattern goes on the same side as the tilde[2] (˜)”.

```
(⋍)   :: (Regex ρ) ⇒ ρ → String → Bool
(˜$)  :: (Regex ρ) ⇒ ρ → String → [Match]
(˜//) :: (Regex ρ, Subst π) ⇒ ρ → String → π
(˜/)  :: (Regex ρ, Subst π) ⇒ ρ → String → π
```

All exported operators are non-associative and bind with priority 4. That makes them bind looser than ++ and :, similar to ≡.

```
infix 4 ≃, ⋍, $˜, ˜$, ˜//, //˜, ˜/, /˜
```

All operators are based on the fundamental pattern matching operation *match*, which is the single method of class Regex:

```
class Regex ρ where
  match :: ρ → String → Maybe Match
```

For the purpose of substitution, functions of a non-polyvariadic type are also provided.

```
subst  :: (Regex ρ) ⇒ ρ → [Repl] → String → String
subst1 :: (Regex ρ) ⇒ ρ → [Repl] → String → String
```

subst performs a global substitution while *subst1* only replaces the first match. Both take the replacement pattern as a list of Repls, representing consecutive parts of the replacement pattern. Each Repl is either a literal replacement string or a submatch reference.

```
data Repl = Repl_lit  String
          | Repl_ref Int
```

Finally, the Match data type is a record containing

1. the substring preceding the match (*m_before*),
2. the matching substring itself (*m_match*),
3. the rest of the string after the match (*m_after*), and
4. the list of strings matching the regex's subexpressions (*m_submatches*).

```
data Match = Match{ m_before      :: String
                  , m_match       :: String
                  , m_after       :: String
                  , m_submatches  :: [String]
                  }
           deriving (Eq, Show, Read)
```

Note that the list of subexpression matches does *not* include the match itself, so for example, *m_submatches* (*head* ("Foo" $˜ "F(o)")) is ["o"], not ["Fo", "o"].

Matching

Compiled regular expressions are represented by the abstract data type CRegex, which wraps Regex from Text.Regex.

```
newtype CRegex = CRegex TR.Regex
```

They are created from regular expression strings by the function *cregex*, which can take options:

[2]In plain text code, ≃ is written as `=˜` and ⋍ as `˜=`, so ≃ is the one taking the pattern on the right.

```
data Rexopt = Nocase | Linematch deriving (Eq, Show, Read)
```
Nocase makes the matching case-insensitive. Linematch results in `'^'` and `'$'` matching start and end of lines instead of the whole string, and `'.'` not matching the newline character. By default, matches are case-sensitive and `'^'` and `'$'` refer to the whole string.

```
cregex :: [Rexopt] → String → CRegex
cregex os s = CRegex (TR.mkRegexWithOpts s lm cs)
  where
  lm = elem Linematch os
  cs  = ¬ (elem Nocase os)
```
The matching operation is overloaded on the regex type. Matching of compiled regexes is performed by a helper *match_cregex*. If the regex is passed as a plain string it is compiled with default options before being passed to *match_cregex*.

```
instance Regex CRegex where
  match = match_cregex
instance Regex String where
  match = match_cregex ∘ cregex [ ]
```
The *match_cregex* function is a wrapper around Text.Regex.*matchRegexAll* whose only purpose is to unwrap the CRegex argument and to wrap the result in a Match.

```
match_cregex :: CRegex → String → Maybe Match
match_cregex (CRegex cr) str =
  do
  (b, m, a, s) ← TR.matchRegexAll cr str
  return $ Match{ m_before       = b
                , m_match        = m
                , m_after        = a
                , m_submatches = s
                }
```
Now, the match testing operators are trivial to define.

```
(⋍) r = isJust ∘ match r
```
I define ≃ in terms of ⋍ and not the other way around, so that applying $(r$ ⋍$)$ to several strings compiles r only once (when r is a string). The same note applies to all other operators as well.

```
(≃)  = flip (⋍)
($~) = flip (~$)
```
The ~$ operator must find all matches within the given string. That can be achieved by consecutively applying *match* to the *m_after* field of the previous match, if any. That's an instance of *unfoldr*.

```
match_all :: (Regex ρ) ⇒ ρ → String → [Match]
match_all r = unfoldr step
  where
  step :: String → Maybe (Match, String)
  step x = do ma ← match r x
              return (ma, m_after ma)
```
This way, however, each match's *m_before* field only extends to the end of the previous match. The list returned by *match_all* is only meaningful in its original order. For ther operators, I expand the matches to span the entire string.

```
(~$) r = expand_matches ∘ match_all r
```

Let m be a match, as retured by *match_all*. If m is the first match in the list, it does not need to be expanded. It's expansion is the empty string `""`. If, on the other hand, m has a predecessor p, its expansion is *m_before p* ++ *m_match p*. So the list of expansions for all matches is given by:

```
expansions :: [Match] → [String]
expansions ms = "" : map (λp → m_before p ++ m_match p) ms
```

That list contains one extraneous entry at the end, but that can be ignored because *expand_matches* is now a simple instance of *zipWith*[3].

```
expand_matches :: [Match] → [Match]
expand_matches ms = zipWith expand ms (expansions ms)
  where
  expand m s = m{ m_before = s ++ m_before m}
```

Substitution

```
class Subst π where
  subst′ :: String → [Match] → [Repl] → π
instance Subst String where
  subst′ s ms rs = replace ms (reverse rs) s
instance (Subst π) ⇒ Subst (String → π) where
  subst′ s ms rs = λx → subst′ s ms (Repl_lit x : rs)
instance (Subst π) ⇒ Subst (Int → π) where
  subst′ s ms rs = λi → subst′ s ms (Repl_ref i : rs)
replace :: [Match] → [Repl] → String → String
replace []            _   s  =  s
replace (m : ms) rs   _   =  (  m_before m
                               ++ concatMap replstr rs
                               ++ replace ms rs (m_after m)
                               )
  where
  replstr r = case r of
        Repl_lit x      → x
        Repl_ref 0      → m_match m
        Repl_ref i      → m_submatches m !! (i − 1)
subst   r = λrs s → replace (match_all r s) rs s
subst1  r = λrs s → replace (take 1 (match_all r s)) rs s
(~//) r = λs → subst′ s (match_all r s) []
(~/) r  = λs → subst′ s (take 1 (match_all r s)) []
(//~) = flip (~//)
(/~)  = flip (~/)
```

[3] Applause!