Обработка текста в Haskell — поиск и замена

Вводные замечания

Поговорим об обработке текста в Haskell, вернее о наиболее востребованных операциях: *поиске* и *замене*.

Модуль `Data.List` предоставляет три функции, которые могут рассматриваться как варианты поиска подстроки в строке:

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool

 > "Hello" `isPrefixOf` "Hello World!"
True
 > "Hello" `isPrefixOf` "Wello Horld!"
False

isSuffixOf :: Eq a => [a] -> [a] -> Bool

 > "ld!" `isSuffixOf` "Hello World!"
True
 > "World" `isSuffixOf` "Hello World!"
False

isInfixOf :: Eq a => [a] -> [a] -> Bool

 > isInfixOf "Haskell" "I really like Haskell."
True
 > isInfixOf "Ial" "I really like Haskell."
False
> isInfixOf "I really" "I really like Haskell."
True
> isInfixOf "ll." "I really like Haskell."
True
```

Фактически, функцией поиска можно назвать последнюю из трёх, но и она не слишком полноценная по сравнению с распространённой функцией **`index`** в различных языках программирования, которая обычно возвращает ещё и индекс позиции найденной подстроки.

Функции замены, вроде `replace` или `substr` в этом модуле нет.

В Haskell есть другой модуль, предоставляющий полезный тип данных `Text` из модуля `Data.Text`, который более экономично и эффективно относится к обработке текстовых строк по сравнению с обычными **`String`**. Возможно, позже мы ещё обсудим его подробнее. А сейчас, посмотрим на его функцию `replace`, которая может делать замену.

```
replace :: Text -> Text -> Text -> Text
```

(«что меняем», «на что», «в какой строке»)

Вот пример работы с этой функцией:

```
import Data.Text(pack, unpack, replace)

replacedoubleslash :: String -> String
```

```
replacedoubleslash s =
   unpack $ replace (pack "//" ) (pack "..") (pack s)

str = "djfjjgfj//djjje"
str2 = replacedoubleslash str
```

и результат:

```
*Main> str2
"djfjjgfj..djjje"
```

Регулярные выражения в Haskell

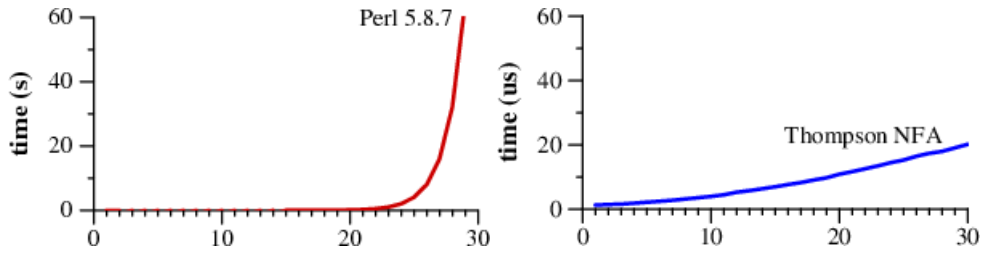
Вводные замечания

Ситуация с регулярными выражениями, и надо признать этот факт сразу, в экосистеме Haskell удручающая. Долгое время их просто не было. В какой-то момент появилась реализация регулярных выражений через библиотеку контекстно-свободных грамматик Parsec, что сразу давало огромное отставание по скорости. Да, они теперь есть, но удобство, документированность, в каких-то аспектах возможность и скорость оставляют желать лучшего. Эта ситуация несравнима с возможностями и удобством использования регулярных выражений в таких языках как Perl, PHP, Ruby, Python, JavaScript или таких инструментах как grep, awk и sed.

Эталонной и самой развитой реализаций до сих пор считается реализация регулярных выражений в Perl. Здесь регулярные выражения являются естественной частью языка, сам их синтаксис и возможности привели к особому стандарту PCRE. Сейчас существует библиотека на Си, которая реализует все возможности PCRE и портируется на другие платформы и языки, написанная Филипом Хейзелом (инспирирована более ранней работой, библиотекой regex, разработанной известным канадским программистом Генри Спенсером).

Но чуть раньше сформировался стандарт POSIX для регулярных выражений. В настоящее время отличия в возможностях не слишком заметны, и многие возможности из PCRE реализованы в POSIX. Однако, некоторые возможности PRCE (так и не вошедшие в POSIX), требуют особой реализации, называемой backtracking, что приводит на некоторых особых примерах к колоссальной потери производительности.

Вот пример такой ситуации (правда довольно искусственный):



Здесь ищется регулярное выражение вида

```
a?a?a?a?a?aaaaa
```

в строке `"aaaaa"`. С ростом n (имеется в виду регулярные выражения $(a?)^n a^n$ в строках a^n, где запись x^n означает повтор x ровно n раз записанным явно образом, т.е. без n) время исполнения на Perl растёт экспоненциально, тогда как время POSIX'овской реализации растёт чуть больше чем линейно (на самом деле в данном случае даже должно $O(n^2)$).

Подробнее об этом читать в статье Regular Expression Matching Can Be Simple And Fast… от Russ Cox (одного из разработчиков языка Golang).

Для примера: на современном ноутбуке с 2 ядрами и процессором i5 при $n = 30$ на поиск ушло 55 сек.

Тем не менее, библиотека PCRE и подход, принятый в Perl являются де-факто промышленным стандартом. И к этому мы ещё вернёмся, говоря о Haskell.

Регулярные выражения в теории

Давайте теперь, рассмотрим что это такое с точки зрения теории.

Определение. Пусть Σ — конечный алфавит, не содержащий символов (,), $\cup$, $*$. Определим по индукции множество регулярных выражений над языком (множеством слов) в алфавите Σ:

1. Множества $\varnothing$, $\{\Lambda\}$, $\{a\}$, где a — произвольный элемент из Σ, являются регулярными выражениями;
2. Если α и β уже являются регулярными выражениями, то $(\alpha\beta)$, $(\alpha \cup \beta)$ и α^* тоже являются регулярными выражениями (иногда вместо $(\alpha \cup \beta)$ пишут $(\alpha \mid \beta)$).

(скобки используются для естественной группировки)

Изначально они были введены математиком-логиком Стивеном Клини в 1951 году, а в конце 60-х была первая «образцово-показательная» реализация Кена Томпсона из Bell Labs в редакторах QED и ed (кстати Томпсон тоже работает в Google над проектированием языка Golang)

У Клини регулярные выражения описывали различные множества слов из данного алфавита. Так, $\varnothing$ описывало пустое множество, Λ — множество из одного пустого слово (пустой строки в обычных языках программирования), т.е. $\{\Lambda\}$, а a — множество $\{a\}$. Более сложные выражения работали следующим образом:

- Конкатенация $(\alpha\beta)$ описывала конкатенацию двух множеств слов, из которых первое описывалось выражением α, а второе — β. При этом конкатенация двух множеств слов L_1 и L_2 определялась следующим образом

$$L_1 L_2 = \{uv \mid u \in L_1, v \in L_2\},$$

т.е. конкатенации множеств слов $L_1 L_2$ — это множество таких слов, у которых первая половина лежит в L_1, а вторая — в L_2.

- Объединение $(\alpha \cup \beta)$ описывало объединение в математическом смысле слов двух множеств, из которых первое описывалось выражением α, а второе — β.

- Звёздочка Клини α^* описывала звёздочку Клини над множеством слов, которое задано выражением α. При этом звёздочка Клини над множеством слов определялась так:

$$L^* = \{w \mid w = w_1 w_2 \dots w_n \text{ для некоторых } n \in \mathbb{N} \text{ и } w_1, w_2, \dots, w_n \in L\},$$

в частности при $n = 0$ получим, что $\Lambda \in L^*$.

ru.wiki: Регулярный язык

Примеры

Для простоты зафиксируем основной алфавит в два символа $\Sigma = \{a, b\}$.

1. Регулярное выражение $abba$ описывает множество, состоящее из одного слова $\{abba\}$.
2. Регулярное выражение $(ba \cup baa)^*$ описывает бесконечное множество слов, состоящее из всевозможных повторов слогов ba и baa в любом порядке и количестве. Множество слов включает в себя и пустое слово, таким образом, получаем:

$$\{\Lambda, ba, baa, baba, bababaa, baaba, \dots\}.$$

3. Регулярное выражение $(b^*ab^*ab^*ab^*)^* \cup b^*$ описывает множество слов, содержащее кратное 3 число букв a (пустое слово содержит 0 букв, что тоже кратно 3).

Но следует отметить, что возможности распознавания (вычислений) регулярных выражений *ограничены в принципе* — так, для такого достаточно простого множества слов $\{a^n b^n \mid n \in \mathbb{N}\}$ не существует регулярного выражения, его описывающего.

Сами регулярные выражения распознаются стандартными алгоритмами, называемыми «конечными автоматами» (eng. Finite-state machine), которые в свою очередь бывают детерминированными и недетерминированными (ДКА и НДКА).

Регулярные выражения на практике

В ИТ-практике мы используем регулярные выражения чаще не для описаний множеств слов, а для составления «развитых» поисковых операций и сопоставления строк (ну или мы ищем описанное множество слов в заданной строке, тексте и т.п.). Таким образом, регулярное выражение `a` означает, например, в соответствующем запросе, есть подстрока `"a"` в заданной строке. Регулярное выражение $(ba \cup baa)$, только теперь мы будем его писать как `(ba|baa)`, описывает наличие подстроки `"ba"` или `"baa"` в заданной строке, т.е. `|` теперь подразумевает выбор (*дизъюнкцию запросов*). А регулярное выражение со звёздочкой Клини `(ba|baa)*` означает повтор ноль или более число раз подряд всевозможных комбинаций строк `"ba"` или `"baa"` в заданной строке.

Следует сказать, что и в POSIX, и тем более в PCRE, выразительные возможности и число различных команд и опций намного больше того, что мы ввели в качестве определения.

Зачем вообще возиться с регулярными выражениями? Чем они могут помочь именно вам?

- **Сравнение с шаблоном:** Регулярные выражения отлично помогают определять, соответствует ли строка тому или иному формату – например, телефонному номеру, адресу электронной почты или номеру кредитной карты.

- **Замена:** При помощи регулярных выражений легко находить и заменять шаблоны в строке. Так, выражение `s/s+/ /g)` заменяет все пробелы в текстовой строке, например, `" \n\t "`, одним пробелом.
- **Извлечение:** При помощи регулярных выражений легко извлекать из шаблона фрагменты информации. Например, `/^(Mr|Ms|Mrs|Dr)/i` извлекает из строки обращение к человеку, например, `"Mr"` из `"Mr. Smith"`.
- **Портируемость:** Почти в любом распространённом языке программирования есть своя библиотека регулярных выражений. Синтаксис в основном стандартизирован, поэтому вам не придётся переучиваться регулярным выражениям при переходе на новый язык.
- **Производительность:** Когда пишете код, можно пользоваться регулярными выражениями для поиска информации в файлах; обычно в IDE для этого предусмотрен **`find and`** `replace`.

(по мотивам Регулярные выражения для простых смертных)

ru.wikipedia: Регулярные выражения

en.wikipedia: Regular expression

ibm.dev: Секреты регулярных выражений

ibm.dev: Часть 2. Регулярные выражения в конкретных программах

wikibooks: Регулярные выражения

Regexp — это «язык программирования». Основы

Квантификация

Так, например, помимо классической звёздочки Клини `*`, определяется операция `+`, что означает повтор один или более число раз (выражения перед плюсом), операция `?`, что означает встречаемость ноль или один число раз. И ещё можно указать точный диапазон повторов:

`{n}`	ровно `n` раз
`{m,n}`	от `m` до `n` раз
`{m,}`	`m` и более раз
`{,n}`	`n` или меньше число раз

Пример. Регулярное выражение `(ba|baa){3,5}` описывает поиск строк вида:

`bababa, babaabaaba, ..., baabaabaabaabaa`

Квантификаторы в регулярных выражениях

Сверхжадные квантификаторы

Представление символов

Обычные символы

Представляют сами себя. Это все символы, за исключением специальных. Можно использовать и управляющий символы `\f \n \r \t \v`.

В Haskell есть поддержка юникода для строк.

Специальные символы

Эти символы

```
[ ] \ / ^ $ . | ? * + ( ) { }
```

несут особую роль в регулярных выражениях и должны быть экранированы с помощью бэкслеша \, если мы хотим использовать в качестве самих «себя» \[или *. В разных стандартах и реализациях это множество может немного отличаться. Например, POSIX-реализация в Haskell не использует в качестве управляющего обратный бэкслеш (его можно использовать как неэкранированный, так и экранированный), но использует : и его надо экранировать.

Рассмотрим функции некоторых из них.

Символ . означает «любой символ». В некоторых реализациях — за исключением перевода строки. Где и как — надо разбираться экспериментально или по документации.

Круглые скобки () помимо группировки ещё несут возможность запомнить результат прямо в этом выражении, или сослаться потом в замене или последующей обработке. Напр., странноватое выражение

```
(па|ма)\1
```

найдёт или слово «папа», или «мама».

В более сложных случаях повторяющихся ()() или вложенных скобок (()) меньший номер имеет объемлющее скобочное выражение. А соседние ()() скобочные выражения дают подряд идущие номера. Например, выражение

```
((aa|oo)(1111|222))\3\2\1
```

опишет строки, подобные такой

```
"aa222222aaaa222"
```

Подсказка:

```
aa.222.222.aa.aa222
 2   3  \3 \2    \1
```

Точками разбито на части, соответственно определяемые частями регулярного выражения. Снизу: числа обозначают номер группы, бэкслэш с числом обозначает использование соответствующей ссылки.

Regex nested backreference

Скобочные группы

Далее, в квадратных скобках [] указываем, что в данном месте может стоять один из перечисленных символов. В частности, [abc] задаёт возможность появления в тексте одного из трёх указанных символов, а [1234567890] задаёт соответствие одной из цифр. Возможно указание диапазонов символов: например, [A-Z] соответствует одна из прописных букв, [a-z] — одна из строчных, а [A-Za-z] — любая из букв латинского алфавита.

Если требуется указать символы, которые не входят в указанный набор, то используют символ `^` внутри квадратных скобок, например `[^0-9]` означает любой символ, кроме цифр.

Сам значок `^`, если его нужно указать, должен идти не первым (также и дефис, если нужен как дефис, а не указатель диапазона, должен быть либо первым, либо последним). Ограничимся для описания латиницей ASCII (должно описывать и локальные кодировки, и unicode). Это небольшое описание спец.символов PCRE:

`\d`	`[0-9]`	Цифровой символ
`\D`	`[^0-9]`	Нецифровой символ
`\s`	`[ \f\n\r\t\v]`	Пробельный символ
`\S`	`[^ \f\n\r\t\v]`	Непробельный символ
`\w`	`[A-Za-z0-9_]`	Букв. или цифр. символ, знак подчёрк
`\W`	`[^A-Za-z0-9_]`	Любой символ, букв/цифр/подчёрк

(пустое место — это тоже пробел)

Спец.символы `\d \D \s \S \w \W` могут использоваться и самостоятельно.

Это небольшое описание символьных классов POSIX (тоже, описание, ограниченное ASCII, но должно работать и в др. кодировках)

`[:upper:]`	`[A-Z]`	Символы верхнего регистра
`[:lower:]`	`[a-z]`	Символы нижнего регистра
`[:alpha:]`	`[[:upper:][:lower:]]`	Буквы
`[:digit:]`	`[0-9]`, т.е. `\d`	Цифры
`[:alnum:]`	`[[:alpha:][:digit:]]`	Буквы и цифры
`[:word:]`	`[[:alnum:]_]`, т.е. `\w`	Символы, образующие «слово»
`[:punct:]`	`[.,;?:…]`	Знаки пунктуации
`[:blank:]`	`[ \t]`	Только пробел и табуляция
`[:space:]`	`[[:blank:]\v\r\n\f]`, т.е. `\s`	Пробельные символы

ru.wiki: Символьные классы POSIX

en.wiki: POSIX Extended Regular Expressions. Character_classes

Позиция

Отметим два важных управляющих символа `^` — начало данной строки и `$` — конец данной строки. Напр., выражение `^Hello` означает `"Hello"` в начале данной строки, а `world$` означает `"Hello"` — в конце.

Полезные ссылки по регулярным выражениям

Фридл Д. *Регулярные выражения*. 3-е изд. Символ-Плюс, 2008.

Фицджеральд М. *Регулярные выражения: основы*. М.: Вильямс, 2015.

Регулярные выражения, пособие для новичков. Часть 1

Регулярные выражения, пособие для новичков. Часть 2

Регулярные выражения для простых смертных

Скобочные группы

30 примеров полезных регулярных выражений

wikipedia: Регулярные выражения

wikibooks: Регулярные выражения

Регулярные выражения в Perl

Регулярные выражения (PCRE)

Диалекты и возможности. Составление регулярных выражений

Шпаргалка по регулярным выражениям

Regular Expressions.info

Regular Expressions (POSIX)

POSIX Basic and Extended Regular Expressions

POSIX-Extended Regular Expressions

Regex Tutorial - Unicode Characters and Properties

Регулярные выражения: никакой магии

Атомарная группировка, или Ни шагу назад!

Comparison of regular-expression engines

RE2 (software)

Практическое использование регулярных выражений в Haskell

С Haskell Platform поставляется библиотека Text.Regex.Posix с помощью которой мы и рассмотрим практическое использование регулярных выражений.

Стоит сказать, что начиная с версии 8.6.3, этот модуль отсутствует в Haskell Platform в варианте *full* и модуль надо устанавливать в пакете `regex-posix` самостоятельно:

```
cabal update
cabal install --lib regex-posix
```

Отметим, что документация для модуля `Text.Regex.Posix` большая, но слишком «бестолковая», чтобы новичку из неё понять, как использовать возможности модуля. Начать, пожалуй, стоит со статьи с примерами

Bryan O'Sullivan. A Haskell regular expression tutorial

которую мы и возьмём в качестве стартовой точки. Некоторые детали можно почитать тут в устаревшей версии модуля Text.Regex.Base.Context v.0.93.2 и в более новой Text.Regex.Base.RegexLike, которые лежат в глубине документации `Text.Regex.Posix`.

Пример 1.

Опишем с помощью регулярного выражения расширение в имени файла:

```
\.txt$|\.doc$
```

Код на Haskell:

```
import Text.Regex.Posix

ext = "\\.txt$|\\.doc$"
filename = "test1.txt"

test = if (filename =~ ext) then "Ok" else "Wrong"
```

Выполнение в ghci:

```
*Main> test
"Ok"
```

Отметим несколько важных моментов здесь.

1. Необходима явная загрузка модуля **`import`** `Text.Regex.Posix`.
2. Паттерн для регулярного выражения при работе с этим модулем в Haskell задаётся с помощью строк **`String`**, и поэтому нам необходимо экранировать бэкслэши (на практике оказалось, что некоторые символы могут быть экранированы и одним бэкслэшем, напр., `\:`, но во избежание путаницы, будем работать как указано).
3. Сравнение с регулярным выражением делается в Perl-стиле с помощью оператора `=~`, при этом задаётся необходимый тип результата. Так, наиболее часто используемый тип результата это булево значение, как в `test`. При необходимости получить более детальную информацию есть возможность использовать разные типы результата (подробности см. в документации). Ниже опишем более экзотичные примеры.
4. Это POSIX-совместимый модуль, Perl-совместимости тут нет. Поэтому, например, если нам нужно искать цифры, используем `[0-9]` или `[:digit:]`, но не используем `\d`.

Опишем вместо

```
test = if (filename =~ ext) then "Ok" else "Wrong"
```

более информативный вывод:

```
test :: String
test = filename =~ ext
```

Тогда мы можем узнать результат нашего поиска:

```
*Main> test
".txt"
```

Пример 2.

Опишем простой валидатор введённого адреса электронной почты, при этом он нам нужен не для реальной проверки, а чтобы посмотреть полиморфизм результата и что он содержит.

```
import Text.Regex.Posix

str = "mister.twister7@gmail.com"
{-
pat = "^([A-Za-z\\._\\d]+)@([A-Za-z_\\d]+)\\.([A-Za-z_\\d]+)$"
  -Perl Style
pat = "^([A-Za-z\\._[:digit:]]+)@([A-Za-z_[:digit:]]+)
      \\.([A-Za-z_[:digit:]]+)$"
 -POSIX
compatible with both versions:
-}
pat = "^([A-Za-z\\._0-9]+)@([A-Za-z_0-9]+)\\.([A-Za-z_0-9]+)$"

test  = str =~ pat :: String
test2 = str =~ pat :: [[String]]
```

Отметим, что Google Mail не разрешает символы подчёркивания в именах, ну собственно проверка корректности адреса майла должна идти либо на основании RFC:

Какие символы разрешены в адресе электронной почты?

wikipedia: Email address

либо более жёсткой проверкой:

Никогда не проверяйте e-mail адреса по стандартам RFC

На 100% правильный способ проверки адресов электронной почты

Продолжим, в ghci получим:

```
*Main> test
"mister.twister7@gmail.com"
*Main> test2
[["mister.twister7@gmail.com","mister.twister7","gmail","com"]]
```

Вывод функции `test` содержит найденную (первую) строку поиска, а более изощрённый результат `test2` выводит ещё содержимое скобочных групп. Таким образом, мы можем извлечь имя пользователя или доменное имя.

Вот простой пример, который более детально показывает работу

```
import Text.Regex.Posix

str = "aa222222aaaa222oo1111"
pat = "(aa|oo)(1111|222)"

test = str =~ pat :: String
test2 = str =~ pat :: [[String]]
```

а в ghci получим:

```
*Main> test
"aa222"
*Main> test2
[["aa222","aa","222"],["aa222","aa","222"],["oo1111","oo","1111"]]
```

т.е. видим, что действительно, функция `test` вывела первую найденную подстроку, а `test2` вывела список всех найденных подстрок и связанные с ними группы.

Пример 2.2

Немного изменим пример: добавим произвольный текст до и после адреса, соответственно изменим регулярное выражение и программный код:

```
import Text.Regex.Posix

str = "We are testing this address: mister.twister@gmail.com!!"
pat = "([A-Za-z\\._0-9]+)@([A-Za-z_\\d]+)\\.([A-Za-z_\\d]+)"

test3 :: (String,String,String,[String])
test3 = str =~ pat

getter (_,x,_,_) = x
getres = getter $ test3
```

В ghci:

```
*Main> test3
("We are testing this address: ","mister.twister@gmail.com","!!",
["mister.twister","gmail","com"])
```

В нашем случае, мы используем тип `(String,String,String,[String])`, который сообщает нам результат-кортеж, содержащий часть слова «до», «найденный результат», «после», найденные группы. Чтобы извлечь результат, используем `getter`:

```
*Main> getres
"mister.twister@gmail.com"
```

Отметим, что пакет `regex-posix`, содержащий модуль `Text.Regex.Posix` производит ощущение бажного и нестабильного. Следующий пример работает не так как надо!

Пример 3.

Простая обработка адреса URL видеоролика на Youtube вида

```
https://www.youtube.com/watch?v=dnOj1EyMLgg
```

с помощью выражения на Haskell:

```
^https\\://www\\.youtube\\.com/watch\\?
```

уже возвращает пустую строку! Баг! Если убрать `^` в начале строки или `s` в протоколе, то работает.

Собственно, пример предполагал вытащить протокол, доменное имя и хэш видеоролика:

```
^(https)://www\.([a-z]+)\.[a-z]+/watch+\?v\=([A-Za-z0-9]+)
```

на Haskell:

```
pat = "^(https)://www\\.([a-z]+)\\.[a-z]+/watch\\?v=([A-Za-z0-9]+)"
test4 = url =~ pat
```

Этот код вообще приводит к крэшу:

```
*Main> test4
"Segmentation fault/access violation in generated code
```

> Теперь эти тесты в Haskell Platform с GHC ver.8.6.3 все проходят, код полностью рабочий. Вопрос с ver. 8.4.2 открытый!

```
import Text.Regex.Posix

pat = "(https)://www\\.([A-Za-z]+)\\.[A-Za-z]+/watch\\?v=([A-Za-z0-9]+)"
url =  "https://www.youtube.com/watch?v=dnOj1EyMLgg"

test = if (url =~ pat) then "Ok" else "Wrong"

test2 :: (String,String,String,[String])
test2 = url =~ pat

test3 :: [[String]]
test3 = url =~ pat

test4 :: String
test4 = url =~ pat
```

и вывод в ghci:

```
*Main> test4
"https://www.youtube.com/watch?v=dnOj1EyMLgg"
*Main> test3
[["https://www.youtube.com/watch?v=dnOj1EyMLgg","https",
  "youtube","dnOj1EyMLgg"]]
*Main> test2
("","https://www.youtube.com/watch?v=dnOj1EyMLgg","",["https",
  "youtube","dnOj1EyMLgg"])
```

Исходя из вышесказанного, нам придётся сменить библиотеку на стабильную, проверенную годами, и имеющую более обширные возможности PCRE:

```
cabal install regex-pcre-builtin
```

или

```
cabal install regex-pcre
```

если у нас в системе уже установлена библиотека PCRE (как это можно сделать в Windows, обсудим позже).

Запустим код на Haskell:

```
import Text.Regex.PCRE

pat = "^(https)://www\\.([A-Za-z]+)\\.[A-Za-z]+/watch\\?v=([A-Za-z\\d]+)"
url =  "https://www.youtube.com/watch?v=TyXs1yzSIKc"

test = if (url =~ pat) then "Ok" else "Wrong"
```

```
test2 :: (String,String,String,[String])
test2 = url =~ pat

test3 :: [[String]]
test3 = url =~ pat

test4 :: String
test4 = url =~ pat
```

и вывод в ghci:

```
*Main> test
"Ok"
*Main> test2
("","https://www.youtube.com/watch?v=TyXs1yzSIKc","",
  ["https","youtube","TyXs1yzSIKc"])
*Main> test3
[["https://www.youtube.com/watch?v=TyXs1yzSIKc","https","youtube",
  "TyXs1yzSIKc"]]
*Main> test4
"https://www.youtube.com/watch?v=TyXs1yzSIKc"
```

Ещё стоит опять отметить не очень хорошую документацию библиотеки, но как уже указывали выше, некоторые детали можно читать тут в устаревшей версии Text.Regex.Base.Context v.0.93.2 и в более новой Text.Regex.Base.RegexLike.

Пример 4.

Следующий пример не работает в POSIX уже принципиально, так как использует ссылки на группы. Поэтому, далее будем использовать библиотеку PCRE.

Простые палидромы. Опишем с помощью регулярного выражения палидромы, состоящие из строчных латинских букв, с длинной слова ровно 5 букв.

```
([a-z])([a-z])[a-z]\2\1
```

И теперь код на Haskell:

```
import Text.Regex.PCRE

str = "djehfabbbashehtr";
pat = "([a-z])([a-z])[a-z]\\2\\1"

test = if (str =~ pat) then "Ok" else "Wrong"

test2 :: (String,String,String,[String])
test2 = str =~ pat

test3 :: [[String]]
test3 = str =~ pat
```

```
test4 :: String
test4 = str =~ pat
```

И в ghci это работает следующим образом:

```
*Main> test
"Ok"
*Main> test2
("djehf","abbba","shehtr",["a","b"])
*Main> test3
[["abbba","a","b"]]
*Main> test4
"abbba"
```

Отметим, что тут существенно использование ссылок на ранее найденные группы, ведь заранее нам не известно, какие группы символов будут найдены!

Пример 5.

О поддержке кириллицы. Она есть, работает:

```
import Text.Regex.Posix

pat = ”([а-я!]+)$”
text = ”Привет, мой друг!”

test = if (text =~ pat) then ”Ok” else ”Wrong”

test4 :: String
test4 = text =~ pat
```

Однако, в ghci с кириллицей плохо (из-за особенности реализации **show**):

```
*Main> test4
"\1076\1088\1091\1075!"
```

Поэтому, делаем так:

```
*Main> putStrLn $ test4
друг!
```

или для «магии» включения кириллицы в ghci, необходимо в начале файла поместить:

```
{-# LANGUAGE FlexibleInstances #-}

instance {-# OVERLAPPING #-} Show String where
    show x = ['"'] ++ x ++ ['"']
```

stackoverflow: Haskell IO with non English characters

Show instance of Char should print literals for non-ascii printable charcters

---

Поиск и замена с помощью регулярных выражений в Haskell

К сожалению, библиотека `Text.Regex.Posix` вообще не предоставляет возможности использования регулярных выражений для поиска с заменой. Поэтому, сразу рассмотрим библиотеку Text.Regex.PCRE.Heavy (она в статусе экспериментальной).

Возможности поиска почти те же самые, что и в библиотеке Text.Regex.PCRE, но добавлен ряд интересных функций: `scan`, `scanRanges`, есть отличия в применении регулярных выражений.

Например, регулярные выражения здесь задаются не строками, а особой техникой (квазицитирование…), поэтому бэкслэши экранировать не надо.

Пример

Вот, кстати, рабочий пример с Ютубом:

```
{-# LANGUAGE QuasiQuotes, FlexibleContexts #-}

import Text.Regex.PCRE.Heavy

url =  "https://www.youtube.com/watch?v=TyXs1yzSIKc"

test =
 if
  url =~
   [re|^(https)://www\.([A-Za-z]+)\.[A-Za-z]+/watch\?v=([A-Za-z\d]+)|]
 then "Ok" else "Wrong"
```

Но главное для нас — это функции, которые дают возможности замены текста.

Пример 1.

```
{-# LANGUAGE QuasiQuotes, FlexibleContexts #-}

import Text.Regex.PCRE.Heavy

str =  "ghdhjjsaaabbfjjrjrjaaaabbbjgjfGHd\
\aabvkk374ktaaabbbbgjjtoYgvvwy"

test = scan [re|(a+)(b+)|] str
```

Зададим «диковинную мульти-линейную» строку `str`, см.

*How can I write multiline strings in Haskell?

Далее, с помощью функции `scan` запустим поиск с шаблоном `(a+)(b+)`. Эта функция сканирует всю исходную строку, и все подстроки, соответствующие шаблону, выводит в виде списка пар, где первый элемент пары показывает очередную найденную подстроку, а второй элемент пары в виде списка показывает содержание всех использованных группировок. Примерно так, как это было в тестах выше с паттерном типа `[[String]]`.

```
*Main> test
[("aaabb",["aaa","bb"]),("aaaabbb",["aaaa","bbb"]),
("aab",["aa","b"]),("aaabbbb",["aaa","bbbb"])]
```

Функция `scan` ленивая, и таким образом, если нам нужно только первый найденный результат, то достаточно использовать **`head`** `$ scan ...`.

Пример 2.

Для начала рассмотрим когда-то популярную проблему, как убрать все тэги в html-файле. Аккуратное решение этой проблемы требует дерево синтаксического разбора. Но мы сделаем «топорное решение в первом приближении» с помощью такого регулярного выражения:

```
<[^<>]+>
```

В этот раз мы будем использовать функцию `gsub`, которая использует теперь 3 аргумента. Первым будет по-прежнему поисковое регулярное выражение, вторым — требуемая замена, третьим — входная строка для обработки.

Вот такой код на Haskell:

```
{-# LANGUAGE QuasiQuotes, FlexibleContexts #-}

import Text.Regex.PCRE.Heavy

str = "<p>We are testing <i>this</i> address: \
\<b>mister.twister@gmail.com</b>, and this one: \
\<a href=\"mailto:hacker@yahoo.com\">hacker@yahoo.com</a>, too.</p>"

str2 = gsub [re|<[^<>]+>|] "" str
```

Который вполне ожидаемо выдаёт:

```
*Main> str2
"We are testing this address: mister.twister@gmail.com,
and this one: hacker@yahoo.com, too."
```

Пример 3.

Замена доменных имён в почтовых адресах:

```
{-# LANGUAGE QuasiQuotes, FlexibleContexts #-}

import Text.Regex.PCRE.Heavy

str = "We are testing this address: mister.twister@gmail.com,\
\ and this one: hacker@yahoo.com, too."

str2 = gsub [re|@[A-Za-z]+\.[A-Za-z]+|] "@mail.ru" str
```

получаем:

```
*Main> str2
"We are testing this address: mister.twister@mail.ru,
and this one: hacker@mail.ru, too."
```

Пример 4.

Изменение имён в почтовых адресах. Мы можем сделать более-менее индивидуальную обработку имён в наших почтовых адресах. Например, так:

```
{-# LANGUAGE QuasiQuotes, FlexibleContexts #-}

import Text.Regex.PCRE.Heavy

str = "We are testing this address: mister.twister@gmail.com, \
\and this one: hacker@yahoo.com, too."

str2 = gsub [re|([A-Za-z.]+)@|] (\(x:_) ->
          (reverse x ++ "@") :: String) str
```

получаем:

```
*Main> str2
"We are testing this address: retsiwt.retsim@gmail.com,
and this one: rekcah@yahoo.com, too."
```

Здесь используется хитрая и опять слабодокументированная техника. Но тем, не менее, разберём выражение

```
\(x:_) -> (reverse x ++ "@") :: String
```

Аннотацию типа `:: String` требует модуль, так как он работает не только со стандартными строками. Далее, мы видим анонимную функцию, которая принимает на вход список групп (в нашем случае, это одна группа), вычленяем с помощью конструктора типа `:` первый элемент (можно было бы и функцию **head** использовать), содержащий нашу группу. Затем, обрабатываем его с помощью функции **reverse** и добавляем знак @). Если бы хотели обработать не группы, а найденное значение, то аргумент был бы не списком строк, а одной строкой.

Если использовать на входе строку, то для достижения того же эффекта придётся сделать так:

```
\x -> ((reverse $ init x) ++ "@") :: String
```

Для сравнения, вот как выглядит эта программа на Perl:

```
$str = 'We␣are␣testing␣this␣address:␣mister.twister@gmail.com,
␣and␣this␣one:␣hacker@yahoo.com,␣too.';
$str =~ s/([A-Za-z.]+)@/reverse($1).'@'/ge;
print $str;
```

Так что тут, мы можем определённо написать:

```
s/Haskell/Perl/;

:( :( :(
```

Ссылки по пакету `pcre-heavy`

Text.Regex.PCRE.Heavy

github: pcre-heavy

haskell regex substitution

Другие полезные ссылки по теме

RegEx101: online tester (для работы в POSIX-стиле надо выбрать `Golang`)

regex-applicative: Regex-based parsing with applicative interface (ещё один пакет на чистом Haskell, с возможностью поиска и замены)

wiki.haskell: Regular expressions (устарело)
