Регулярні вирази широко використовуються в багатьох скриптових мовах таких як Perl, Python або Ruby. Вони корисні для того щоб визначити чи стрічка відповідає певному шаблону, перевірити чи вхідні дані відповідають очікуваному формат, і щоб обробити стрічку в новий формат. Pharo також підтримує регулярні вирази завдяки пакету Regex, який додав Василь Биков. Regex пакет встановлений в Pharo за замовчуванням. Якщо ви використовуєте давнішу версію яка не включає в себе пакет Regex, ви можете встановити його самостійно з SqueakSource (<http://www.squeaksource.com/Regex.html>).

[Регулярний вираз](http://en.wikipedia.org/wiki/Regular_expression) - це рядок, що описує або збігається з множиною рядків. Для прикладу регулярний вираз'h.\*o' будуть задовільняти стрічки 'ho', 'hiho' і 'hello', але не стрічки 'hi' or 'yo'. Ми можемо спостерігати це у Pharo на прикладах:

'ho' matchesRegex: 'h.\*o'

>>> **true**

'hiho' matchesRegex: 'h.\*o'

>>> **true**

'hello' matchesRegex: 'h.\*o'

>>> **true**

'hi' matchesRegex: 'h.\*o'

>>> **false**

'yo' matchesRegex: 'h.\*o'

>>> **false**

У цьому розділі ми почнемо з маленького прикладу, в якому ми розробимо декілька класів щоб згенерувати простеньку карту сайту для веб сайту. Ми використаємо регулярні вирази:

1. для того щоб ідентифікувати HTML файли,
2. для того щоб отримати з шляху до файлу назву файлу,
3. для того щоб отримати заголовок кожної веб сторінки для карти сайту, і
4. для того щоб згенерувати відносний шлях від кореневої директорії веб сайту до HTML файлів які вона містить.

Після того як ми завершимо цей приклад, ми наведемо біль повний опис пакету Regex, який в основному базується на документації Василя Бикова, яка надається в пакеті. (Оригінальна документація може буде знайдена на стороні класу RxParser.)

1. Приклад — генерування карти сайту

Наша робота полягає в тому щоб написати простеньку програму яка буде генерувати карту сайту для веб сайту який зберігається локально на жорсткому диску. Карта сайту буде містити посилання на кожну з HTML сторінок на веб сайті, використовуючи заголовок документа як текст посилання. Крім того, посилання будуть вирівняні відступами для того щоб відображати структуру директорій веб сайту.

1.1. Доступ до веб директорії

Якщо у вас немає веб сайту на вашому комп'ютері, скопіюйте декілька HTML файлів, які будуть слугувати як тестові дані, в локальну директорію

Ми розробимо два класи, WebDir і WebPage, які будуть представляти директорії і веб сторінки відповідно. Ідея в тому щоб створити екземпляр класу WebDir, який буде вказувати на кореневу директорію, яка містить на веб сайт. Коли ми відправимо цьому екземпляру повідомлення makeToc, він пройде по всіх файлах і директоріях всередині даної, щоб збудувати карту сайту. Потім він створить новий вайл, з назвою toc.html, який буде містити посилання на всі сторінки в веб сайті.

Єдина річ за якою потрібно стежити: кожен екземпляр класів WebDir і WebPage має пам'ятати шлях до кореневої директорії веб сайту, для того щоб генерувати правильні посилання відносно до кореня.

Визначити клас WebDir зі змінними рівня екземпляру webDir і homePath, і визначити відповідний метод для ініціалізації.

Також визначити метод рівня класу, який буде отримувати від користувача розташування веб сайту на комп'ютері:

**WebDir** >> setDir: dir home: path

webDir := dir.

homePath := path

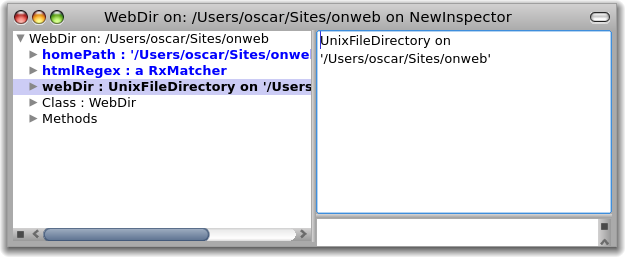
**WebDir** class >> onDir: dir

^ **self** new setDir: dir home: dir pathString

**WebDir** class >> selectHome

^ **self** onDir: **UIManager** default chooseDirectory

Останній метод відкриває файловий браузер щоб вибрати яку директорію відкрити. Тепер, якщо ми дослідимо результат WebDir selectHome, вам буде запропоновано вибрати директорію яка містить ваші веб сторінки, і ви зможете переконатись, що webDir і homePath правильно проініціалізовані директорією що містить ваш веб сайт, і повним шляхом до цієї директорії.

0.1. A WebDir instance.

Було б добре мати змогу програмно встановити WebDir, тож давайте добавимо інший метод для створення.

Додайте наступні методи, і спробуйте їх виконати, глянувши на результат WebDir onPath: 'шлях до вашого веб сайту'.

**WebDir** class>>onPath: homePath

^ **self** onPath: homePath home: homePath

**WebDir** class>>onPath: path home: homePath

^ **self** new setDir: path asFileReference home: homePath

1.2. Шаблон який відповідає HTML файлам

Тепер давайте використаємо регулярні вирази для того щоб з'ясувати які HTML файли цей веб сайт містить.

Якщо ми переглянемо клас FileDirectory, ми вияснимо, що метод fileNames повертає список всіх файлів в директорії. Нам потрібно вибрати тільки ті, які мають розширення .html. Регулярний вираз, який нам необхідний це '.\*\.html'. Перша крапка відповідає будь якому символу, крім символу нової стрічки:

'x' matchesRegex: '.'

>>> **true**

' ' matchesRegex: '.'

>>> **true**

**Character** cr asString matchesRegex: '.'

>>> **true**

Символ \* (Також відомий як *Зірка Кліні*, яку придумав Стівен Кліні) - це оператор регулярного виразу, який відповідає попередньому виразу будь-яку кількість раз (включаючи нуль).

'' matchesRegex: 'x\*'

>>> **true**

'x' matchesRegex: 'x\*'

>>> **true**

'xx' matchesRegex: 'x\*'

>>> **true**

'y' matchesRegex: 'x\*'

>>> **false**

Оскільки крапка це спеціальний символ в регулярних вираз, то якщо ми хочемо щоб стрічка буквально збігалась з крапкою, то ми маємо екранувати крапку. (*.*).

'.' matchesRegex: '.'

>>> **true**

'x' matchesRegex: '.'

>>> **true**

'.' matchesRegex: '\.'

>>> **true**

'x' matchesRegex: '\.'

>>> **false**

Тепер давайте перевіримо що наш регулярний вираз для пошуку HTML файлів працює як очікувалось

'index.html' matchesRegex: '.\*\.html'

>>> **true**

'foo.html' matchesRegex: '.\*\.html'

>>> **true**

'style.css' matchesRegex: '.\*\.html'

>>> **false**

'index.htm' matchesRegex: '.\*\.html'

>>> **false**

Виглядає добре. Тепер давайте спробуємо його в нашій програмі.

Добавте наступний метод до класу WebDir і спробуйте протестувати його на вашому веб сайті.

**WebDir** >> htmlFiles

^ webDir fileNames select: [ :each | each matchesRegex: '.\*\.html' ]

Якщо надіслати повідомлення htmlFiles до екземпляру WebDir і використати опцію print it в випадаючому списку, ви повинні бачити щось на зразок цього:

(**WebDir** onPath: '...') htmlFiles

>>> #('index.html' ...)

1.3. Кешування регулярного виразу

Тепер, якщо ви переглянете результати matchesRegex:, ви з'ясуєте що це метод розширення класу String, який створює новий екземпляр класу RxParser кожен раз як надсилається повідомлення. Це допустимо для спеціальних запитів, але якщо ми застосовуємо той же регулярний вираз для кожного файлу в веб сайті, то краще було б створити тільки один екземпляр класу RxParser, і повторно використовувати його. Давайте зробимо це.

Додати нову змінну рівня екземпляру htmlRegex до класу WebDir і ініціалізувати її, відправляючи повідомлення asRegex до стрічки з регулярним виразом. Модифікувати метод WebDir>>htmlFiles для використання одного і того ж регулярного виразу:

**WebDir** >> initialize

htmlRegex := '.\*\.html' asRegex

**WebDir** >> htmlFiles

^ webDir fileNames select: [ :each | htmlRegex matches: each ]

Тепер отримання списку HTML файлів повинно працювати так же як перед тим, за винятком того що ми повторно використовуємо той же екземпляр декілька раз.

1.4. Доступ до веб сторінок

Доступ до деталей окремих веб сторінок - повинно бути відповідальністю окремого класу, то ж давайте визначимо його, і нехай клас WebDir буде створювати екземпляри.

Визначити клас WebPage зі змінними рівня екземпляру path, для ідентифікації HTML файлу, і homePath, для ідентифікації кореневої директорії веб сайту. (Нам буде необхідно це для того щоб правильно генерувати посилання з кореневої директорії веб сайту до файлів які вона містить.) Визначте метод для ініціалізації на рівні екземпляру, і метод для створення на рівні класу.

**WebPage** >> initializePath: filePath homePath: dirPath

path := filePath.

homePath := dirPath

**WebPage** class >> on: filePath forHome: homePath

^ **self** new initializePath: filePath homePath: homePath

Екземпляр класу WebDir має повертати список всіх веб сторінок які містить ця директорія.

Додайте наступний метод до класу WebDir, і впевніться що він працює.

**WebDir** >> webPages

^ **self** htmlFiles collect:

[ :each | **WebPage**

on: webDir pathString, '/', each

forHome: homePath ]

Ви повинні спостерігати щось на зразок цього:

(**WebDir** onPath: '...') webPages

>>> an **Array**(a **WebPage** a **WebPage** ...)

1.5. Підстановка стрічок

Це не дуже інформативно, то ж давайте використаємо регулярний вираз для того щоб отримати фактичну назву файлу для кожної веб сторінки. Для цього на потрібно стерти всі символи зі шляху, аж до останньої директорії. В файловій системі операційних систем типу UNIX, назви директорій завершуються символом «слеш» (/), тому нам необхідно видалити все аж до останнього «слешу» в шляху до файлу.

Метод розширення класу String який називається copyWithRegex:matchesReplacedWith: робить те що нам потрібно:

'hello' copyWithRegex: '[elo]+' matchesReplacedWith: 'i'

>>> 'hi'

В цьому випадку регулярний вираз [elo] відповідає будь якому з символів e, l або o. Оператор + майже ідентичний до Зірки Кліні, за винятком того що він відповідає точно *одному* або більше екземпляру виразу що передує йому. Тут він відповідає цілій підстрічці 'ello' і заміняє її літерою i.

Добавимо наступний метод, і переконаємось що він працює як очікувалось.

**WebPage** >> fileName

^ path copyWithRegex: '.\*/' matchesReplacedWith: ''

Тепер ви повинні бачити щось на зразок цього на вашому тестовому веб сайті:

(**WebDir** onPath: '...') webPages collect: [:each | each fileName ]

>>> #('index.html' ...)

1.6. Отримання результатів збігів з регулярним виразом

Нашим наступним завданням є отримання заголовків кожної HTML сторінки. Для початку нам потрібно отримати вміст кожної сторінки. Це досить просто.

Добавте наступний метод і виконайте його.

**WebPage** >> contents

^ (**FileStream** oldFileOrNoneNamed: path) contents

Насправді, ви можете мати проблеми якщо ваші веб сторінки містять не ASCII символи, в цьому випадку краще використати наступний код:

**WebPage** >> contents

^ (**FileStream** oldFileOrNoneNamed: path)

converter: **Latin1TextConverter** new;

contents

Тепер ви повинні бачити щось на зразок цього:

(**WebDir** onPath: '...') webPages first contents

>>> '<head>

<title>Home Page</title>

...

'

Тепер давайте отримаємо заголовок. В цьому випадку ми шукаємо текст, який знаходиться *між* HTML тегами <title> і </title>.

Що нам необхідно, це спосіб витягнути *частину* збігу з регулярним виразом. Підвирази регулярних виразів розділяються дужками. Розглянемо наступний регулярний вираз ([CARETaeiou]+)([aeiou]+); він складається з двох підвиразів, перший з яких буде збігатись з послідовністю однієї або більше неголосних літер, і другий буде збігатись з послідовністю однієї або більше голосних літер: оператор CARET на початку множини в квадратних дужках, заперечує цю множину. (Зверніть увагу: в Pharo символ каретки є також ключовим словом return, яке ми записуємо також як ^. Щоб уникнути плутанини, ми будемо записувати CARET, коли використовуємо каретку в регулярному виразі щоб заперечити множину символів, але не треба забувати, що це насправді одне й те ж.) Тепер ми будемо намагатись отримати *префікс* стрічки 'pharo' і отримати збіги:

| re |

re := '([CARETaeiou]+)([aeiou]+)' asRegex.

re matchesPrefix: 'pharo'

>>> **true**

re subexpression: 1

>>> 'pha'

re subexpression: 2

>>> 'ph'

re subexpression: 3

>>> 'a'

Після успішного співставлення регулярного виразу зі стрічкою, ви завжди можете надіслати повідомлення subexpression: 1 для того щоб отримати повний збіг. Ви також можете надіслати повідомлення subexpression: n, де n-1 - це число підвиразів в регулярному виразі. Регулярний вираз зверху має два підвирази, які нумеруються як 2 і 3.

Ми використаємо цей самий спосіб що витягнути заголовок з HTML файла.

Визначте наступний метод:

**WebPage** >> title

| re |

re := '[\w\W]\*<title>(.\*)</title>' asRegexIgnoringCase.

^ (re matchesPrefix: **self** contents)

ifTrue: [ re subexpression: 2 ]

ifFalse: [ '(', **self** fileName, ' -- untitled)' ]

Є декілька тонких моментів які треба тут відзначити. По-перше, розмітка HTML не вимагає щоб теги були в верхньому чи нижньому регістрі, тому ми маємо зробити наш регулярний вираз не чутливим до регістру створюючи його за допомогою asRegexIgnoringCase.

По-друге, оскільки крапка збігається з будь-яким символом *крім символу нової стрічки*, регулярний вираз .\*<title>(.\*)</title> не буде працювати правильно якщо декілька стрічок передують заголовку. Вираз \w збігається з будь-яким алфавітно-цифровим символом, і \W збігається з будь яким не алфавітно-цифровим символом, тому [\w\W] буде збігатись з будь яким символом *включаючи символ нової стрічки*. (Якщо нам потрібно щоб заголовки могли містити символи нової стрічки, то потрібно використати цей самий спосіб для підвиразу.)

Тепер можна протестувати наш метод для отримання заголовку, і ми повинні побачити щось на зразок цього:

(**WebDir** onPath: '...') webPages first title

>>> 'Home page'

1.7. Більше підстановки стрічок

Для того щоб згенерувати карту сайту, нам необхідно згенерувати посилання на окремі веб сторінки. Ми можемо використати заголовок документа як ім'я посилання. Нам необхідно просто згенерувати правильний шлях до веб сторінки з кореневої директорії веб сайту. На щастя, це досить тривіально - потрібно просто відняти шлях до кореневої директорії веб сайту від повного шляху до веб сторінки.

Нам потрібно остерігатись тільки однієї речі. Оскільки, змінна homePath не закінчується на /, то нам потрібно додати його, так щоб відносний шлях не включав в себе передуючого символу /. Зауважте різницю між наступними результатами:

'/home/testweb/index.html' copyWithRegex: '/home/testweb' matchesReplacedWith: ''

>>> '/index.html'

'/home/testweb/index.html' copyWithRegex: '/home/testweb/' matchesReplacedWith: ''

>>> 'index.html'

Перший результат дає нам абсолютний шлях, що ймовірно не те що ми хочемо.

Визначте наступні методи:

**WebPage** >> relativePath

^ path

copyWithRegex: homePath, '/'

matchesReplacedWith: ''

**WebPage** >> link

^ '<a href="', **self** relativePath, '">', **self** title, '</a>'

Тепер ви повинні бачити щось на зразок цього:

(**WebDir** onPath: '...') webPages first link

>>> '<a href="index.html">Home Page</a>'

1.8. Генерування карти сайту

Насправді, на цьому етапі ми закінчили з регулярними виразами, які необхідні для того щоб згенерувати карту сайту. Нам лише необхідно ще декілька методів щоб завершити нашу програму.

Якщо ви хочете побачити генерування карти сайту, вам необхідно просто добавити наступні методи.

Якщо наш веб сайт має підкаталоги, нам необхідно спосіб щоб доступитись до них:

**WebDir** >> webDirs

^ webDir directoryNames

collect: [ :each | **WebDir** onPath: webDir pathString, '/', each home: homePath ]

Нам необхідно згенерувати маркований список в HTML який буде містити посилання на кожну веб сторінку веб директорії. Піддерикторії повинні бути зображені з відступом в їхньому власному маркованому списку.

**WebDir** >> printTocOn: aStream

**self** htmlFiles

ifNotEmpty: [

aStream nextPutAll: '<ul>'; cr.

**self** webPages

do: [:each | aStream nextPutAll: '<li>';

nextPutAll: each link;

nextPutAll: '</li>'; cr].

**self** webDirs

do: [:each | each printTocOn: aStream].

aStream nextPutAll: '</ul>'; cr]

Ми створюємо файл під назвою *toc.html* в кореневій директорії скидаємо карту сайту тут.

**WebDir** >> tocFileName

^ 'toc.html'

**WebDir** >> makeToc

| tocStream |

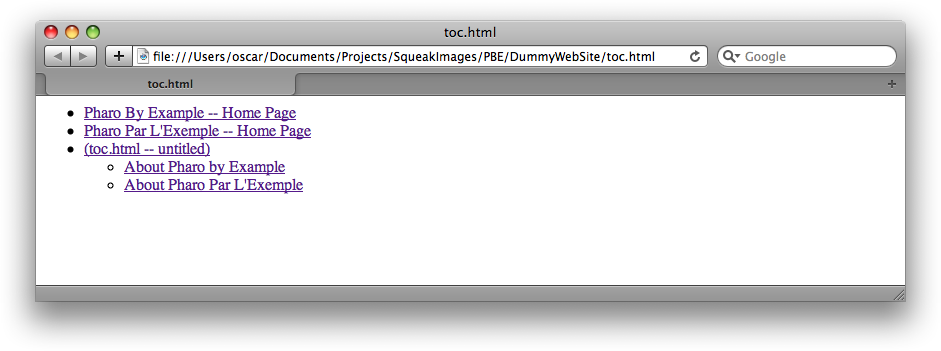
tocStream := (webDir / **self** tocFileName) writeStream.

**self** printTocOn: tocStream.

tocStream close.

Тепер генеруємо зміст для будь якої веб директорії!

**WebDir** selectHome makeToc

0.2. Маленька карта сайту.

2. Синтаксис регулярних виразів

Тепер нам необхідно більш детальний погляд на синтаксис регулярних виразів які підтримує пакет Regex.

Найпростіший регулярний вираз це єдиний символ. Він буде збігатись точно з цим символом. Послідовність символів збігається точно з такою ж послідовністю символів:

'a' matchesRegex: 'a'

>>> **true**

'foobar' matchesRegex: 'foobar'

>>> **true**

'blorple' matchesRegex: 'foobar'

>>> **false**

Оператори застосовуються до регулярних виразів для того щоб створити більш складний регулярний вираз. Послідовність (розташування виразів один за одним) як оператор є, в деякому сенсі, *невидимим* - хоча й, можливо, найпоширенішим.

Ми вже бачили Зірку Кліні (\*) і оператор +. Регулярний вираз за яким слідує зірочка збігається з будь-якою кількістю (включаючи нуль) збігів з початковим виразом. Для прикладу:

'ab' matchesRegex: 'a\*b'

>>> **true**

'aaaaab' matchesRegex: 'a\*b'

>>> **true**

'b' matchesRegex: 'a\*b'

>>> **true**

'aac' matchesRegex: 'a\*b'

>>> **false** "b не збігається"

Зірка Кліні має більший пріорітет ніж послідовність. Зірка застосовується до найкоротшого підвиразу що передує їй. Для прикладу, ab\* означає символ a за яким слідує нуль або більше входжень символу b, але не *нуль або більше входжень ab*:

'abbb' matchesRegex: 'ab\*'

>>> **true**

'abab' matchesRegex: 'ab\*'

>>> **false**

Для того щоб отримати регулярний вираз який збігається з *нулем або більше входжень ab*, ми маємо взяти підвираз ab у дужки:

'abab' matchesRegex: '(ab)\*'

>>> **true**

'abcab' matchesRegex: '(ab)\*'

>>> **false** "c портить збіг"

Два інші корисні оператори схожі на \* є + і ?. + збігається з одним або більше входженням виразу який він модифікує, і ? збігається з нулем або одним входженням.

'ac' matchesRegex: 'ab\*c'

>>> **true**

'ac' matchesRegex: 'ab+c'

>>> **false** "потребує хоча б одне b"

'abbc' matchesRegex: 'ab+c'

>>> **true**

'abbc' matchesRegex: 'ab?c'

>>> **false** "забагато символів b"

Як ми бачимо, символи \*, +, ?, (, і ) мають спеціальне значення в регулярному виразі. Якщо нам потрібні буквальні збіги з ними, ми повинні екранувати їх використовуючи символ «бекслеш» (\ ) перед ними. Таким чином «бекслеш» також являється спеціальним символом, і потребує екранування для буквального збігу. Те ж саме справедливо і для наступних спеціальних символів які ми побачимо.

'ab\*' matchesRegex: 'ab\*'

>>> **false** "зірочка справа - це спеціальний символ"

'ab\*' matchesRegex: 'ab\\*'

>>> **true**

'a\c' matchesRegex: 'a\\c'

>>> **true**

Останній оператор - це |, який виражає вибір між двома підвиразами. Він збігається зі стрічкою якщо хоча б один з двох підвиразів збігається зі стрічкою. Він має найнижчий пріорітет - навіть нижчий ніж послідовність. Для прикладу, ab\*|ba\* означає *символ a за яким слідує будь-яка кількість символів b, або символ b за яким слідує будь-яка кількість символів a*:

'abb' matchesRegex: 'ab\*|ba\*'

>>> **true**

'baa' matchesRegex: 'ab\*|ba\*'

>>> **true**

'baab' matchesRegex: 'ab\*|ba\*'

>>> **false**

Більш складний приклад це вираз c(a|d)+r, який збігається з ім'ям будь-якої з функцій car, cdr, caar, cadr в стилі мови Lisp:

'car' matchesRegex: 'c(a|d)+r'

>>> **true**

'cdr' matchesRegex: 'c(a|d)+r'

>>> **true**

'cadr' matchesRegex: 'c(a|d)+r'

>>> **true**

Можна записати вираз який збігається з пустою стрічкою, для прикладу вираз a| збігається з пустою стрічкою. Однак, застосовувати \*, +, або ? до такого виразу: (a|)\* є помилкою.

До сих пір ми використовували символи як *найменші* компоненти регулярного виразу. Хоча, є інші, більш цікаві, компоненти. Множина символів це стрічка оточена квадратними дужками. Вона збігається з будь-яким символом якщо він знаходиться між дужками. Для прикладу, [01] збігається або з 0 або з 1:

'0' matchesRegex: '[01]'

>>> **true**

'3' matchesRegex: '[01]'

>>> **false**

'11' matchesRegex: '[01]'

>>> **false** "множина збігається тільки з одним символомі"

Використовуючи оператор +, ми можемо записати наступний розпізнавач бінарних чисел:

'10010100' matchesRegex: '[01]+'

>>> **true**

'10001210' matchesRegex: '[01]+'

>>> **false**

Якщо першим символом після відкриваючої квадратної дужки є CARET, то множина інвертується: вона збігається з будь-яким символом, який *не* знаходиться між дужками:

'0' matchesRegex: '[CARET01]'

>>> **false**

'3' matchesRegex: '[CARET01]'

>>> **true**

Для зручності, множина може включати діапазони: пари символів розділені дефісом (-). Це еквівалент до перечислювання всіх символів які знаходяться між даними: '[0-9]' це те ж саме що '[0123456789]'. Спеціальні символи в множині це CARET, -, і ], яка закриває множину. Дальше подані приклади як отримати буквальний збіг з ними в множині:

'CARET' matchesRegex: '[01CARET]'

>>> **true** "поставити CARET де-небудь крім початку"

'-' matchesRegex: '[01-]'

>>> **true** "поставити дефіс в кінці"

']' matchesRegex: '[]01]'

>>> **true** "поставити закриваючу дужку на початку"

Таким чином, пусті і універсальні множини не можуть бути визначені.

2.1. Класи символів

Регулярні вирази також можуть включати наступні символи для позначення популярних класів символів: \w відповідає будь-яким алфавітно-цифровим символам, \d відповідає цифрам, і \s відповідає пробільними символами. Їхні варіанти в верхньому регістрі, \W, \D і \S, використовуються для відповідно протилежних множин (не алфавітно-цифрових, не цифр, і не пробільних символів). Далі поданий короткий виклад синтаксису який ми бачили до сих пір:

|  |  |
| --- | --- |
| **Синтаксис** | **Що він представляє** |
| a | буквальний збіг з символом a |
| . | збіг з будь-яким символом (крім символу нової стрічки) |
| (...) | груповий підвираз |
| \x | екранувати наступний спеціальний символ, де 'x' може набувати значень: 'w','s','d','W','S','D' |
| \* | Зірка Кліні — збіг з попереднім підвиразом нуль або більше раз |
| + | збіг з попереднім підвиразом один або більше раз |
| ? | збіг з попереднім підвиразом нуль або один раз |
| | | збіг з одним з підвиразів зліва або справа |
| [abcd] | збіг з будь-яким з символів abcd |
| [^abcd] | збіг з будь-яким з символів не в множині abcd |
| [0-9] | збіг з діапазоном символів від 0 до 9 |
| \w | відповідає алфавітно-цифровим символам |
| \W | відповідає не алфавітно-цифровим символам |
| \d | відповідає цифрам |
| \D | відповідає не цифрам |
| \s | відповідає пробільним символам |
| \S | відповідає не пробільним символам |

Як зазначено у вступі, регулярні вирази особливо корисні для перевірки вводу користувача, і класи символів виявляються особливо корисними для визначення таких регулярних виразів. Для прикладу, невід'ємні числа відповідають регулярному виразу \d+:

'42' matchesRegex: '\d+'

>>> **true**

'-1' matchesRegex: '\d+'

>>> **false**

Ще краще, ми могли б сказати що ненульові числа не повинні починатись з цифри 0:

'0' matchesRegex: '0|([1-9]\d\*)'

>>> **true**

'1' matchesRegex: '0|([1-9]\d\*)'

>>> **true**

'42' matchesRegex: '0|([1-9]\d\*)'

>>> **true**

'099' matchesRegex: '0|([1-9]\d\*)'

>>> **false** "передуючий 0"

Ми також можемо перевірити на додатні і від'ємні числа:

'0' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> **true**

'-1' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> **true**

'42' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> **true**

'+99' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> **true**

'-0' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> **false** "від'ємний zero"

'01' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> **false** "передуючий zero"

Числа з плаваючою комою повинні вимагати принаймні одної цифри після крапки:

'0' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> **true**

'0.9' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> **true**

'3.14' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> **true**

'-42' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> **true**

'2.' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> **false** "потребує цифр після символу ."

На десерт, ось розпізнавач числового формату в загальному: будь-що як 999, або 999.999, або -999.999e+21.

'-999.999e+21' matchesRegex: '(\+|-)?\d+(\.\d\*)?((e|E)(\+|-)?\d+)?'

>>> **true**

Класи символів також можуть включати наступні елементи які сумісні з утилітою grep:

|  |  |
| --- | --- |
| **Синтаксис** | **Що він представляє** |
| [:alnum:] | будь-які алфавітно-цифрові символи |
| [:alpha:] | будь-які алфавітні символи |
| [:cntrl:] | будь-які керуючі символи (з кодом ascii нижче 32) |
| [:digit:] | будь-які десяткові цифри |
| [:graph:] | будь-які графічні символи (з кодом ascii вище 32) |
| [:lower:] | будь-які символи нижнього регістру |
| [:print:] | будь-які символи які можна надрукувати (тут те ж саме, що [:graph:]) |
| [:punct:] | будь-які символи пунктуації |
| [:space:] | будь-які пробільні символи |
| [:upper:] | будь-які символи верхнього регістру |
| [:xdigit:] | будь-які шістнадцяткові цифри |

Зауважте, що ці елементи є компонентами класів символів, *тобто* вони мають бути оточені додатковою парою квадратних дужок щоб сформувати правильний регулярний вираз. Для прикладу, регулярний вираз який представляє стрічки з непустих цифр: [[:digit:]]+. Зазначені вище примітивні вирази і оператори є поширеними в багатьох реалізаціях регулярних виразів.

'42' matchesRegex: '[[:digit:]]+'

>>> **true**

2.2. Класи спеціальних символів

Наступний примітивний вираз унікальний для цієї реалізації Smalltalk. Послідовність символів між двокрапками трактується як унарний селектор який повинен розпізнаватись символами. Символ збігається з таким регулярним виразом якщо він повертає true в відповідь на повідомлення надіслане цим селектором. Це приводить до більш читабельного і ефективного способу визначення класів символів. Для прикладу, [0-9] є еквівалентним до :isDigit:, але наступний більш ефективний. Аналогічно до множини символів, класи символів можуть бути заперечені: :CARETisDigit: збігається з символами які повертають false в відповідь на повідомлення isDigit, і тому є еквівалентним до [CARET0-9].

До сих пір ми побачили наступні еквівалентні способи записати регулярний вираз який збігається з непустою стрічкою чисел: [0-9]+, \d+, [\d]+, [[:digit:]]+, :isDigit:+.

'42' matchesRegex: '[0-9]+'

>>> **true**

'42' matchesRegex: '\d+'

>>> **true**

'42' matchesRegex: '[\d]+'

>>> **true**

'42' matchesRegex: '[[:digit:]]+'

>>> **true**

'42' matchesRegex: ':isDigit:+'

>>> **true**

2.3. Збіг з межами стрічки

Остання група спеціальних примітивних виразів, яка продемонстрована далі, використовується щоб визначати межі стрічки.

|  |  |
| --- | --- |
| **Синтаксис** | **Що він представляє** |
| CARET | будь-яка пуста стрічка на початку рядка |
| \$ | будь-яка пуста стрічка в кінці рядка |
| \b | будь-яка пуста стрічка в межах слова |
| \B | будь-яка пуста стрічка не в межах слова |
| \< | будь-яка пуста стрічка на початку слова |
| \> | будь-яка пуста стрічка в кінці слова |

'hello world' matchesRegex: '.\*\bw.\*'

>>> **true** "межа слова перед w"

'hello world' matchesRegex: '.\*\bo.\*'

>>> **false** "мемає межі слова перед o"

3. Regex API

До сих пір ми зосереджувались в основному на синтаксисі регулярних виразів. Тепер зосередимо увагу на різних типах повідомлень, які розуміють стрічки і регулярні вирази.

3.1. Збіг з префіксами і ігнорування регістру

До сих пір в більшості наших прикладів ми використовували метод розширення matchesRegex: класу String.

Стрічки також розуміють наступні повідомлення prefixMatchesRegex:, matchesRegexIgnoringCase: і prefixMatchesRegexIgnoringCase:.

Повідомлення prefixMatchesRegex: діє просто як matchesRegex, за винятком того що отримувач не повинен повністю збігатись з регулярним виразом переданим як аргумент; збіг тільки префіксу є достатнім.

'abacus' matchesRegex: '(a|b)+'

>>> **false**

'abacus' prefixMatchesRegex: '(a|b)+'

>>> **true**

'ABBA' matchesRegexIgnoringCase: '(a|b)+'

>>> **true**

'Abacus' matchesRegexIgnoringCase: '(a|b)+'

>>> **false**

'Abacus' prefixMatchesRegexIgnoringCase: '(a|b)+'

>>> **true**

3.2. Інтерфейс Enumeration

Деякі аплікації потребують доступу до *усіх* збігів певного регулярного виразу в стрічці. Збіги доступні використовуючи протокол схожий з вже знайомим нам Collection протоколом для перерахування.

regex:matchesDo: обчислює одноаргументний блок для кожного збігу регулярного виразу в рядку отримувача.

| list |

list := **OrderedCollection** new.

'Jack meet Jill' regex: '\w+' matchesDo: [:word | list add: word].

list

>>> an **OrderedCollection**('Jack' 'meet' 'Jill')

regex:matchesCollect: обчислює одноаргументний блок для кожного збігу регулярного виразу в рядку отримувача. Потім збирає результати і повертає їй як екземпляр класу SequenceableCollection.

'Jack meet Jill' regex: '\w+' matchesCollect: [:word | word size]

>>> an **OrderedCollection**(4 4 4)

allRegexMatches: повертає колекцію всіх збігів (підстрічок стрічки отримувача) регулярного виразу.

'Jack and Jill went up the hill' allRegexMatches: '\w+'

>>> an **OrderedCollection**('Jack' 'and' 'Jill' 'went' 'up' 'the' 'hill')

3.3. Заміна і переміщення

Також можливо замінити всі збіги регулярного виразу з певною стрічкою використовуючи повідомлення copyWithRegex:matchesReplacedWith:.

'Krazy hates Ignatz' copyWithRegex: '\<[[:lower:]]+\>' matchesReplacedWith: 'loves'

>>> 'Krazy loves Ignatz'

Біль загальна заміна - це переміщення збігу. Це повідомлення обчислює блок передаючи йому кожен збіг регулярного виразу в стрічці і відповідає копією стрічки з блоком результатів вставлених в неї на місце збігів.

'Krazy loves Ignatz' copyWithRegex: '\b[a-z]+\b' matchesTranslatedUsing: [:each | each asUppercase]

>>> 'Krazy LOVES Ignatz'

Всі повідомлення енумерації і протоколи підстановки виконують збіги чутливі до регістру. Версії не чутливі до регістру не передбачені протоколом класу String. Замість цього вони доступні використовуючи біль низькорівневий інтерфейс для збігів, який представлений в наступній секції.

3.4. Низькорівневий інтерфейс

Коли ви посилаєте повідомлення matchesRegex: до стрічки, стається наступне:

* Створюється новий екземпляр класу RxParser, і йому передається стрічка з регулярним виразом, створюючи синтаксичне дерево регулярного виразу.
* Синтаксичне дерево передається як параметр для ініціалізації до класу RxMatcher. Екземпляр створює структуру даних яка працюватиме як Співставляч регуляного виразу описаного деревом.
* Початкова стрічка передається цьому співставлячу, і він перевіряє стрічку на збіги.

3.5. Співставляч

Якщо ви неодноразово перевіряєте на збіги деяку кількість стрічок з одним і тим же регулярним виразом використовуючи одне з повідомлень яке визначене в класі String, стрічка з регулярним виразом розпізнається і новий співставляч створюється для кожного збігу. Ви можете уникнути цих накладних витрат будуючи співставляч для регулярного виразу, і потім повторно використовуючи його знову і знову. Ви можете, для прикладу, створити співставляч на рівні класу, або на рівні ініціалізації екземпляру, і зберігати його в змінній для повторного використання. Ви можете створити співставляч використовуючи один з наступних методів:

* Ви можете надіслати повідомлення asRegex або asRegexIgnoringCase до стрічки.
* Ви можете напряму викликати конструктор класу RxMatcher використовуючи методи forString: або forString:ignoreCase: (що зроблять зручні методі зазначені вище).

Тут ми посилаємо повідомлення matchesIn: щоб зібрати всі збіги знайдені в стрічці:

| octal |

octal := '8r[0-9A-F]+' asRegex.

octal matchesIn: '8r52 = 16r2A'

>>> an **OrderedCollection**('8r52')

| hex |

hex := '16r[0-9A-F]+' asRegexIgnoringCase.

hex matchesIn: '8r52 = 16r2A'

>>> an **OrderedCollection**('16r2A')

| hex |

hex := **RxMatcher** forString: '16r[0-9A-Fa-f]+' ignoreCase: true.

hex matchesIn: '8r52 = 16r2A'

>>> an **OrderedCollection**('16r2A')

3.6. Співставлення

Співставляч розуміє ці повідомлення (всі вони повертають true що означає успішний збіг, або false в інакшому випадку):

matches: aString — true якщо вся стрічка, яка прийшла аргументом (aString) збігається.

'\w+' asRegex matches: 'Krazy'

>>> **true**

matchesPrefix: aString — true якщо деякий префікс стрічки, що прийшла аргументом (не обов'язково цілої стрічки) збігається.

'\w+' asRegex matchesPrefix: 'Ignatz hates Krazy'

>>> **true**

search: aString — шукає в стріці перше входження підстрічки яка збігається з регулярним виразом. Зауважте, що перші два методи намагаються шукати збіги тільки з самого початку стрічки. Використовуючи приклад зазначений вище з співставлячем для a+, цей метод поверне true для стрічки 'baaa', хоча для попередніх двох буде невдача.

'\b[a-z]+\b' asRegex search: 'Ignatz hates Krazy'

>>> **true** "знаходить 'hates'"

Співставляч також зберігає результат останнього збігу і може повідомити його: lastResult повертає Boolean: результат останньої спроби співставлення. Якщо спроб співставлення ще не було зроблено, то відповідь невизначена.

| number |

number := '\d+' asRegex.

number search: 'Ignatz throws 5 bricks'.

number lastResult

>>> **true**

matchesStream:, matchesStreamPrefix: і searchStream: діють аналогічно до вище зазначених трьох повідомлень, але беруть потоки, як їхні аргументи.

| ignatz names |

ignatz := **ReadStream** on: 'Ignatz throws bricks at Krazy'.

names := '\<[A-Z][a-z]+\>' asRegex.

names matchesStreamPrefix: ignatz

>>> **true**

3.7. Співставлення підвиразів

Після успішної спроби співставлення, ви можете зробити запит на те яка частина початкової стрічки збіглась з якою частиною регулярного виразу. Підвирази це оточена дужками частина регулярного виразу, або цілого виразу. Коли регулярний вираз компілюється, його підвиразам присвоюються індекси від 1, спочатку вглиб, зліва направо.

Для прикладу, регулярний вираз ((\d+)\s\*(\w+)) має чотири підвирази, включаючи себе.

1: ((\d+)\s\*(\w+)) "повний підвираз"

2: (\d+)\s\*(\w+) "найвищий підвираз оточений дужками"

3: \d+ "перший листковий підвираз"

4: \w+ "другий листковий підвираз"

Найбільший можливий індекс рівний одиниці плюс кількість дужок. (Тому, 1 завжди дійсний індекс, навіть якщо немає підвиразів оточених дужками.)

Після успішного збігу, співставляч може відповісти яка частина початкової стрічки відповідає якому підвиразу. Він розуміє ці повідомлення:

subexpressionCount повертає повну кількість підвиразів: найбільше можливе значення яке може бути використане як індекс підвиразу для цього співставляча. Це значення доступне зразу після ініціалізації і ніколи не змінюється.

subexpression: приймає допустимий індекс як аргумент, і може бути надіслане тільки після успішного збігу. Метод повертає підстрічку початкової стрічки яка відповідає з якою відповідний підвираз збігся.

subBeginning: і subEnd: повертає позиції в стрічці або потоці що прийшли аргументом де збіг з даним підвиразом почався і закінчився відповідно.

| items |

items := '((\d+)\s\*(\w+))' asRegex.

items search: 'Ignatz throws 1 brick at Krazy'.

items subexpressionCount

>>> 4

items subexpression: 1

>>> '1 brick' "повний вираз"

items subexpression: 2

>>> '1 brick' "верхній підвираз"

items subexpression: 3

>>> '1' "перший листковий підвираз"

items subexpression: 4

>>> 'brick' "другий листковий підвираз"

items subBeginning: 3

>>> 14

items subEnd: 3

>>> 15

items subBeginning: 4

>>> 16

items subEnd: 4

>>> 21

Як більш складний приклад, наступна частина коду використовує розпізнавач дати в форматі MMM DD, YYYY для того щоб конвертувати дату в три вимірний масив з стрічками року, місяця і дня відповідно:

| date result |

date := '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+(\d\d?)\s\*,\s\*19(\d\d)' asRegex.

result := (date matches: 'Aug 6, 1996')

ifTrue: [{ (date subexpression: 4) .

(date subexpression: 2) .

(date subexpression: 3) } ]

ifFalse: ['no match'].

result

>>> #('96' 'Aug' '6')

3.8. Enumeration і заміна

Протоколи Enumeration і заміни класу String які ми бачили раніше в цій секції реалізовані співставлячем.RxMatcher реалізує наступні методи для проходу по збігах в стрічках: matchesIn:, matchesIn:do:,matchesIn:collect:, copy:replacingMatchesWith: і copy:translatingMatchesUsing:.

| seuss aWords |

seuss := 'The cat in the hat is back'.

aWords := '\<([^aeiou]|[a])+\>' asRegex. "match words with 'a' in them"

aWords matchesIn: seuss

>>> an **OrderedCollection**('cat' 'hat' 'back')

aWords matchesIn: seuss collect: [:each | each asUppercase ]

>>> an **OrderedCollection**('CAT' 'HAT' 'BACK')

aWords copy: seuss replacingMatchesWith: 'grinch'

>>> 'The grinch in the grinch is grinch'

aWords copy: seuss translatingMatchesUsing: [ :each | each asUppercase ]

>>> 'The CAT in the HAT is BACK'

Також існують наступні методі для ітеруваня по збігах всередині потоків: matchesOnStream:,matchesOnStream:do:, matchesOnStream:collect:, copyStream:to:replacingMatchesWith: andcopyStream:to:translatingMatchesUsing:.

3.9. Обробка помилок

Декілька винятків можуть бути підняті класом RxParser при побудові регулярного виразу. Винятки мають спільного батька, клас RegexError. Ви можете використовувати звичний механізм обробки помилок мови Smalltalk для того щоб ловити і обробляти їх.

* RegexSyntaxError кидається якщо синтаксична помилка виявлена при обробці
* RegexCompilationError кидається коли виникає помилка при побудові співставляча.
* RegexMatchingError кидається коли виникає помилка при співставленні (наприклад,

коли некоректний селектор був визначений використовуючи синтаксис ':<selector>:').

['+' asRegex] on: **RegexError** do: [:ex | ^ ex printString ]

>>> 'RegexSyntaxError: nullable closure'

4. Примітки щодо реалізації від Василя Бикова

a. На що звертати увагу перше.

В 90% випадків, метод String>>matchesRegex: це все що потрібно для доступу до пакету.

RxParser приймає стрічку або потік символів з регулярним виразом, і повертає синтаксичне дерево, яке відповідає виразу. Дерево будується з екземплярів класів вигляду Rxs\*.

RxMatcher приймає синтаксичне дерево з регулярного виразу побудованого обробником і компілює його в співставляч: структура побудована з екземплярів класів Rxm\*. Екземпляр класу RxMatcher перевіряє чи стрічка або позиційний потік символів відповідає початковому регулярному виразу, або він може здійснити пошук стрічки чи потоку на підстрічки які відповідають виразу. Після того як збіг знайдений, співставляч може повертати певну стрічку яка збіглась з цілим виразом, або будь яким підвиразом оточеним дужками. Всі інші класу підтримують ту ж функціональній і використовуються класом RxParser або RxMatcher, або обома зразу.

b. Застереження

Співставляч близький по духу, але *не є* реалізований відповідно до початкової реалізації регулярних виразів в мові C Генрі Спенсера. Основна увага приділяється простоті, не ефективності. Я нічого не оптимізовував чи профілював. Співставляч проходить набір тестів Г. Спенсера (дивіться на протокол *test suite*), з декількома тестами доданими, тому є шанси що багів небагато. Але будьте пильними в будь-якому випадку.

c. Вдячність

З першого релізу співставляча, дякуючи допомозі декількох побратимів по Smalltalk, Я переконаний що вбудований співставляч регулярних виразів в Smalltalk був вартий тих зусиль. Для поради і заохочення що зробили цей реліз можливим, я хочу подякувати наступним людям: Felix Hack, Eliot Miranda, Robb Shecter, David N. Smith, Francis Wolinski і всім кого я не зустрічав, і від кого не було ніяких звісток, але хто погоджується що це не було даремною тратою часу.

5. Підсумок глави

Регулярні вирази це істотний інструмент для маніпулювання стрічками тривіальним способом. Ця глава представляє пакет Regex для Pharo. Основними пунктами цієї глави є:

* Для простих збігів, просто надсилайте matchesRegex: то екземпляру класу String
* Коли важлива швидкодія, надсилайте asRegex до стрічки яка представляє регулярний вираз, і повторно використовуєте результуючий співставляч
* Підвирази збігів з регулярним виразом можуть бути легко отримані на довільну глибину
* Регулярний вираз може також заміняти або переміщувати підвирази в нову копію співставленої стрічки
* Інтерфейс Enumeration надається для того щоб доступитись до всіх збігів певного регулярного виразу.
* Регулярний вираз працює як з потоками так і зі стрічками.