Chapter 6

Regular Expressions in Pharo

with the participation of:
Oscar Nierstrasz *(oscar.nierstrasz@acm.org)*

Regular expressions are widely used in many scripting languages such as Perl, Python and Ruby. They are useful to identify strings that match a certain pattern, to check that input conforms to an expected format, and to rewrite strings to new formats. Pharo also supports regular expressions due to the *Regex* package contributed by Vassili Bykov. Regex is installed by default in Pharo.

A regular expression[1] is a template that matches a set of strings. For example, the regular expression 'h.*o' will match the strings 'ho', 'hiho' and 'hello', but it will not match 'hi' or 'yo'. We can see this in Pharo as follows:

```
'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
```

In this chapter we will start with a small tutorial example in which we will develop a couple of classes to generate a very simple site map for a web site. We will use regular expressions (i) to identify HTML files, (ii) to strip the full path name of a file down to just the file name, (iii) to extract the title of each web page for the site map, and (iv) to generate a relative path from the root directory of the web site to the HTML files it contains. After we complete the tutorial example, we will provide a more complete description of the Regex package, based largely on Vassili Bykov's documentation

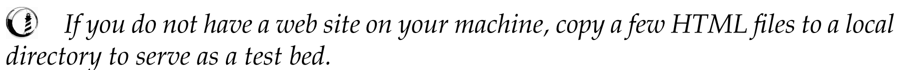
[1] http://en.wikipedia.org/wiki/Regular_expression

provided in the package.[2]

6.1 Tutorial example — generating a site map

Our job is to write a simple application that will generate a site map for a web site that we have stored locally on our hard drive. The site map will contain links to each of the HTML files in the web site, using the title of the document as the text of the link. Furthermore, links will be indented to reflect the directory structure of the web site.

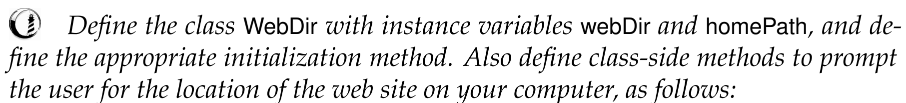
Accessing the web directory



If you do not have a web site on your machine, copy a few HTML files to a local directory to serve as a test bed.

We will develop two classes, WebDir and WebPage, to represent directories and web pages. The idea is to create an instance of WebDir which will point to the root directory containing our web site. When we send it the message makeToc, it will walk through the files and directories inside it to build up the site map. It will then create a new file, called toc.html, containing links to all the pages in the web site.

One thing we will have to watch out for: each WebDir and WebPage must remember the path to the root of the web site, so it can properly generate links relative to the root.



Define the class WebDir *with instance variables* webDir *and* homePath, *and define the appropriate initialization method. Also define class-side methods to prompt the user for the location of the web site on your computer, as follows:*

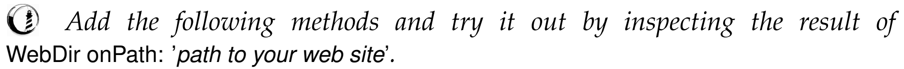
```
WebDir>>setDir: dir home: path
   webDir := dir.
   homePath := path

WebDir class>>onDir: dir
   ^ self new setDir: dir home: dir pathName

WebDir class>>selectHome
   ^ self onDir: FileList modalFolderSelector
```

The last method opens a browser to select the directory to open. Now, if you inspect the result of WebDir selectHome, you will be prompted for the directory containing your web pages, and you will be able to verify that

[2]The original documentation can be found on the class side of RxParser.

webDir and homePath are properly initialized to the directory holding your web site and the full path name of this directory.

It would be nice to be able to programmatically instantiate a WebDir, so let's add another creation method.


Add the following methods and try it out by inspecting the result of WebDir onPath: '*path to your web site*'.

```
WebDir class>>onPath: homePath
   ^ self onPath: homePath home: homePath

WebDir class>>onPath: path home: homePath
   ^ self new setDir: (path asFileReference) home: homePath
```

Pattern matching HTML files

So far so good. Now we would like to use regexes to find out which HTML files this web site contains.

If we browse the AbstractFileReference class, we find that the method fileNames will list all the files in a directory. We want to select just those with the file extension .html. The regex that we need is '.*\.html'. The first dot will match any character.

```
'x' matchesRegex: '.'  ⟶  true
' ' matchesRegex: '.'  ⟶  true
Character cr asString matchesRegex: '.'  ⟶  true
```

The * (known as the "Kleene star", after Stephen Kleene, who invented it) is a regex operator that will match the preceding regex any number of times (including zero).

```
'' matchesRegex: 'x*'  ⟶  true
'x' matchesRegex: 'x*'  ⟶  true
'xx' matchesRegex: 'x*'  ⟶  true
'y' matchesRegex: 'x*'  ⟶  false
```

Since the dot is a special character in regexes, if we want to literally match a dot, then we must escape it.

```
'.' matchesRegex: '.'  ⟶  true
'x' matchesRegex: '.'  ⟶  true
'.' matchesRegex: '\.'  ⟶  true
'x' matchesRegex: '\.'  ⟶  false
```

Now let's check our regex to find HTML files works as expected.

```
'index.html' matchesRegex: '.*\.html'   ⟶   true
'foo.html' matchesRegex: '.*\.html'     ⟶   true
'style.css' matchesRegex: '.*\.html'    ⟶   false
'index.htm' matchesRegex: '.*\.html'    ⟶   false
```

Looks good. Now let's try it out in our application.

Add the following method to WebDir *and try it out on your test web site.*

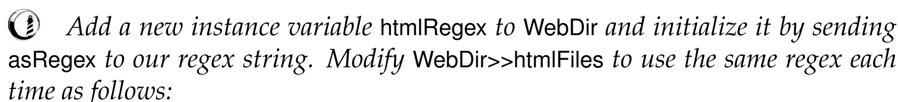
```
WebDir>>htmlFiles
  ^ webDir fileNames select: [ :each | each matchesRegex: '.*\.html' ]
```

If you send htmlFiles to a WebDir instance and print it, you should see something like this:

```
(WebDir onPath: '...') htmlFiles   ⟶   #('index.html' ...)
```

Caching the regex

Now, if you browse matchesRegex:, you will discover that it is an extension method of String that creates a fresh instance of RxParser every time it is sent. That is fine for ad hoc queries, but if we are applying the same regex to every file in a web site, it is smarter to create just one instance of RxParser and reuse it. Let's do that.



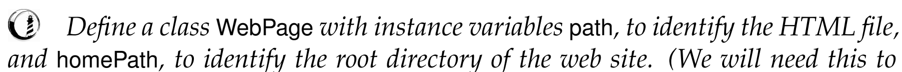
Add a new instance variable htmlRegex *to* WebDir *and initialize it by sending* asRegex *to our regex string. Modify* WebDir>>htmlFiles *to use the same regex each time as follows:*

```
WebDir>>initialize
  htmlRegex := '.*\.html' asRegex

WebDir>>htmlFiles
  ^ webDir fileNames select: [ :each | htmlRegex matches: each ]
```

Now listing the HTML files should work just as it did before, except that we reuse the same regex object many times.

Accessing web pages

Accessing the details of individual web pages should be the responsibility of a separate class, so let's define it, and let the WebDir class create the instances.

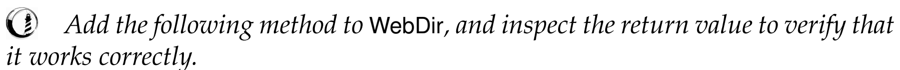


Define a class WebPage *with instance variables* path, *to identify the HTML file, and* homePath, *to identify the root directory of the web site. (We will need this to*

correctly generate links from the root of the web site to the files it contains.) Define an initialization method on the instance side and a creation method on the class side.

```
WebPage>>initializePath: filePath homePath: dirPath
   path := filePath.
   homePath := dirPath

WebPage class>>on: filePath forHome: homePath
   ^ self new initializePath: filePath homePath: homePath
```

A WebDir instance should be able to return a list of all the web pages it contains.



Add the following method to WebDir*, and inspect the return value to verify that it works correctly.*

```
WebDir>>webPages
   ^ self htmlFiles collect:
      [ :each | WebPage
         on: webDir fullName, '/', each
         forHome: homePath ]
```

You should see something like this:

```
(WebDir onPath: '...') webPages    ⟶    an Array(a WebPage a WebPage ...)
```

String substitutions

That's not very informative, so let's use a regex to get the actual file name for each web page. To do this, we want to strip away all the characters from the path name up to the last directory. On a Unix file system directories end with a slash (/), so we need to delete everything up to the last slash in the file path.

The String extension method copyWithRegex:matchesReplacedWith: does what we want:

```
'hello' copyWithRegex: '[elo]+' matchesReplacedWith: 'i'    ⟶    'hi'
```

In this example the regex [elo] matches any of the characters e, l or o. The operator + is like the Kleene star, but it matches exactly *one* or more instances of the regex preceding it. Here it will match the entire substring 'ello' and replay it in a fresh string with the letter i.

Add the following method and verify that it works as expected.

```
WebPage>>fileName
   ^ path copyWithRegex: '.*/' matchesReplacedWith: ''
```

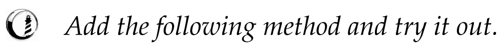
Now you should see something like this on your test web site:

```
(WebDir onPath: '...') webPages collect: [:each | each fileName ]
    ⟶   #('index.html' ...)
```

Extracting regex matches

Our next task is to extract the title of each HTML page.

First we need a way to get at the contents of each page. This is straightforward.



Add the following method and try it out.

```
WebPage>>contents
   ^ (FileStream oldFileOrNoneNamed: path) contents
```

Actually, you might have problems if your web pages contain non-ascii characters, in which case you might be better off with the following code:

```
WebPage>>contents
   ^ (FileStream oldFileOrNoneNamed: path)
      converter: Latin1TextConverter new;
      contents
```

You should now be able to see something like this:

```
(WebDir onPath: '...') webPages first contents    ⟶    '<head>
<title>Home Page</title>
...
'
```

Now let's extract the title. In this case we are looking for the text that occurs *between* the HTML tags <title> and </title>.

What we need is a way to extract *part* of the match of a regular expression. Subexpressions of regexes are delimited by parentheses. Consider the regex ([ˆaeiou]+)([aeiou]+). It consists of two subexpressions, the first of which will match a sequence of one or more non-vowels, and the second of which will match one or more vowels. (The operator ˆ at the start of a bracketed set of characters negates the set. [3])

[3]NB: In Pharo the caret is also the return keyword, which we write as ^. To avoid confusion, we will write ˆ when we are using the caret within regular expressions to negate sets of characters, but you should not forget, they are actually the same thing.

Now we will try to match a *prefix* of the string 'pharo' and extract the sub-matches:

```
re := '([^aeiou]+)([aeiou]+)' asRegex.
re matchesPrefix: 'pharo'    →   true
re subexpression: 1          →   'pha'
re subexpression: 2          →   'ph'
re subexpression: 3          →   'a'
```

After successfully matching a regex against a string, you can always send it the message subexpression: 1 to extract the entire match. You can also send subexpression: n where $n - 1$ is the number of subexpressions in the regex. The regex above has two subexpressions, numbered 2 and 3.

We will use the same trick to extract the title from an HTML file.

Define the following method:

```
WebPage>>title
  | re |
  re := '[\w\W]*<title>(.*)</title>' asRegexIgnoringCase.
  ^ (re matchesPrefix: self contents)
    ifTrue: [ re subexpression: 2 ]
    ifFalse: [ '(', self fileName, ' -- untitled)' ]
```

As HTML does not care whether tags are upper or lower case, so we must make our regex case insensitive by instantiating it with asRegexIgnoringCase.

Now we can test our title extractor, and we should see something like this:

```
(WebDir onPath: '...') webPages first title   →   'Home page'
```

More string substitutions

In order to generate our site map, we need to generate links to the individual web pages. We can use the document title as the name of the link. We just need to generate the right path to the web page from the root of the web site. Luckily this is trivial — it is simple the full path to the web page minus the full path to the root directory of the web site.

We must only watch out for one thing. Since the homePath variable does not end in a /, we must append one, so that relative path does not include a leading /. Notice the difference between the following two results:

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

The first result would give us an absolute path, which is probably not what we want.

Define the following methods:

```
WebPage>>relativePath
  ^ path
    copyWithRegex: homePath , '/'
    matchesReplacedWith: ''

WebPage>>link
  ^ '<a href="', self relativePath, '">', self title, '</a>'
```

You should now be able to see something like this:

```
(WebDir onPath: '...') webPages first link    ⟶    '<a href="index.html">Home Page</a>'
```

Generating the site map

Actually, we are now done with the regular expressions we need to generate the site map. We just need a few more methods to complete the application.

If you want to see the site map generation, just add the following methods.

If our web site has subdirectories, we need a way to access them:

```
WebDir>>webDirs
  ^ webDir directoryNames
    collect: [ :each | WebDir onPath: webDir pathName , '/' , each home: homePath ]
```

We need to generate HTML bullet lists containing links for each web page of a web directory. Subdirectories should be indented in their own bullet list.

```
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]
```

We create a file called “toc.html” in the root web directory and dump the site map there.

```
WebDir>>tocFileName
   ^ 'toc.html'

WebDir>>makeToc
   | tocStream |
   tocStream := (webDir / self tocFileName) writeStream.
   self printTocOn: tocStream.
   tocStream close.
```

Now we can generate a table of contents for an arbitrary web directory!

```
WebDir selectHome makeToc
```

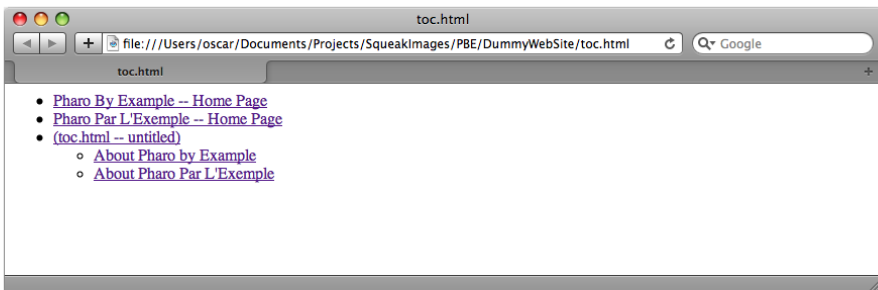


Figure 6.1: A small site map

6.2 Regex syntax

We will now have a closer look at the syntax of regular expressions as supported by the Regex package.

The simplest regular expression is a single character. It matches exactly that character. A sequence of characters matches a string with exactly the same sequence of characters:

```
'a' matchesRegex: 'a'              ⟶  true
'foobar' matchesRegex: 'foobar'    ⟶  true
'blorple' matchesRegex: 'foobar'   ⟶  false
```

Operators are applied to regular expressions to produce more complex regular expressions. Sequencing (placing expressions one after another) as an operator is, in a certain sense, "invisible"—yet it is arguably the most common.

We have already seen the Kleene star (*) and the + operator. A regular expression followed by an asterisk matches any number (including 0) of matches of the original expression. For example:

```
'ab' matchesRegex: 'a*b'        ⟶ true
'aaaaab' matchesRegex: 'a*b'    ⟶ true
'b' matchesRegex: 'a*b'         ⟶ true
'aac' matchesRegex: 'a*b'       ⟶ false  "b does not match"
```

The Kleene star has higher precedence than sequencing. A star applies to the shortest possible subexpression that precedes it. For example, ab* means a followed by zero or more occurrences of b, not "zero or more occurrences of ab":

```
'abbb' matchesRegex: 'ab*'  ⟶ true
'abab' matchesRegex: 'ab*'  ⟶ false
```

To obtain a regex that matches "zero or more occurrences of ab", we must enclose ab in parentheses:

```
'abab' matchesRegex: '(ab)*'   ⟶ true
'abcab' matchesRegex: '(ab)*'  ⟶ false  "c spoils the fun"
```

Two other useful operators similar to * are + and ?. + matches one or more instances of the regex it modifies, and ? will match zero or one instance.

```
'ac' matchesRegex: 'ab*c'     ⟶ true
'ac' matchesRegex: 'ab+c'     ⟶ false  "need at least one b"
'abbc' matchesRegex: 'ab+c'   ⟶ true
'abbc' matchesRegex: 'ab?c'   ⟶ false  "too many b's"
```

As we have seen, the characters *, +, ?, (, and) have special meaning within regular expressions. If we need to match any of them literally, it should be escaped by preceding it with a backslash \. Thus, backslash is also special character, and needs to be escaped for a literal match. The same holds for all further special characters we will see.

```
'ab*' matchesRegex: 'ab*'    ⟶ false  "star in the right string is special"
'ab*' matchesRegex: 'ab\*'   ⟶ true
'a\c' matchesRegex: 'a\\c'   ⟶ true
```

The last operator is |, which expresses choice between two subexpressions. It matches a string if either of the two subexpressions matches the string. It has the lowest precedence — even lower than sequencing. For example, ab*|ba* means "a followed by any number of b's, or b followed by any number of a's":

```
'abb' matchesRegex: 'ab*|ba*'  ⟶ true
```

```
'baa' matchesRegex: 'ab*|ba*'     ⟶  true
'baab' matchesRegex: 'ab*|ba*'    ⟶  false
```

A bit more complex example is the expression c(a|d)+r, which matches the name of any of the Lisp-style car, cdr, caar, cadr, ... functions:

```
'car' matchesRegex: 'c(a|d)+r'    ⟶  true
'cdr' matchesRegex: 'c(a|d)+r'    ⟶  true
'cadr' matchesRegex: 'c(a|d)+r'   ⟶  true
```

It is possible to write an expression that matches an empty string, for example the expression a| matches an empty string. However, it is an error to apply *, +, or ? to such an expression: (a|)* is invalid.

So far, we have used only characters as the *smallest* components of regular expressions. There are other, more interesting, components. A character set is a string of characters enclosed in square brackets. It matches any single character if it appears between the brackets. For example, [01] matches either 0 or 1:

```
'0' matchesRegex: '[01]'    ⟶  true
'3' matchesRegex: '[01]'    ⟶  false
'11' matchesRegex: '[01]'   ⟶  false  "a set matches only one character"
```

Using plus operator, we can build the following binary number recognizer:

```
'10010100' matchesRegex: '[01]+'   ⟶  true
'10001210' matchesRegex: '[01]+'   ⟶  false
```

If the first character after the opening bracket is ˆ, the set is inverted: it matches any single character *not* appearing between the brackets:

```
'0' matchesRegex: '[ˆ01]'   ⟶  false
'3' matchesRegex: '[ˆ01]'   ⟶  true
```

For convenience, a set may include ranges: pairs of characters separated by a hyphen (−). This is equivalent to listing all characters in between: '[0−9]' is the same as '[0123456789]'. Special characters within a set are ˆ, −, and], which closes the set. Below are examples how to literally match them in a set:

```
'ˆ' matchesRegex: '[01ˆ]'    ⟶  true   "put the caret anywhere except the start"
'−' matchesRegex: '[01−]'    ⟶  true   "put the hyphen at the end"
']' matchesRegex: '[]01]'    ⟶  true   "put the closing bracket at the start"
```

Thus, empty and universal sets cannot be specified.

Syntax	What it represents
a	literal match of character a
.	match any char
(· · ·)	group subexpression
\	escape following special character
*	Kleene star — match previous regex zero or more times
+	match previous regex one or more times
?	match previous regex zero times or once
|	match choice of left and right regex
[abcd]	match choice of characters abcd
[^abcd]	match negated choice of characters
[0-9]	match range of characters 0 to 9
\w	match alphanumeric
\W	match non-alphanumeric
\d	match digit
\D	match non-digit
\s	match space
\S	match non-space

Table 6.1: Regex Syntax in a Nutshell

Character classes

Regular expressions can also include the following backquote escapes to refer to popular classes of characters: \w to match alphanumeric characters, \d to match digits, and \s to match whitespace. Their upper-case variants, \W, \D and \S, match the complementary characters (non-alphanumerics, non-digits and non-whitespace). We can see a summary of the syntax seen so far in Table 6.1.

As mentioned in the introduction, regular expressions are especially useful for validating user input, and character classes turn out to be especially useful for defining such regexes. For example, non-negative numbers can be matched with the regex d+:

```
'42' matchesRegex: '\d+'   ⟶   true
'−1' matchesRegex: '\d+'   ⟶   false
```

Better yet, we might want to specify that non-zero numbers should not start with the digit 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  "leading 0"
```

We can check for negative and positive numbers as well:

```
'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  "negative zero"
'01' matchesRegex: '(0|((\+|−)?[1−9]\d*))'    ⟶  false  "leading zero"
```

Floating point numbers should require at least one digit after the dot:

```
'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  "need digits after ."
```

For dessert, here is a recognizer for a general number format: anything like 999, or 999.999, or −999.999e+21.

```
'−999.999e+21' matchesRegex: '(\+|−)?\d+(\.\d*)?((e|E)(\+|−)?\d+)?'  ⟶  true
```

Character classes can also include the grep(1)-compatible elements listed in Table 6.2.

Syntax	What it represents
[:alnum:]	any alphanumeric
[:alpha:]	any alphabetic character
[:cntrl:]	any control character (ascii code is < 32)
[:digit:]	any decimal digit
[:graph:]	any graphical character (ascii code >= 32)
[:lower:]	any lowercase character
[:print:]	any printable character (here, the same as [:graph:])
[:punct:]	any punctuation character
[:space:]	any whitespace character
[:upper:]	any uppercase character
[:xdigit:]	any hexadecimal character

Table 6.2: Regex character classes

Note that these elements are components of the character classes, *i.e.*, they have to be enclosed in an extra set of square brackets to form a valid regular expression. For example, a non-empty string of digits would be represented as [[:digit:]]+. The above primitive expressions and operators are common to many implementations of regular expressions.

```
'42' matchesRegex: '[[:digit:]]+'    ⟶    true
```

Special character classes

The next primitive expression is unique to this Smalltalk implementation. A sequence of characters between colons is treated as a unary selector which is supposed to be understood by characters. A character matches such an expression if it answers true to a message with that selector. This allows a more readable and efficient way of specifying character classes. For example, [0−9] is equivalent to :isDigit:, but the latter is more efficient. Analogously to character sets, character classes can be negated: :^isDigit: matches a character that answers false to isDigit, and is therefore equivalent to [^0−9].

So far we have seen the following equivalent ways to write a regular expression that matches a non-empty string of digits: [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
```

Matching boundaries

The last group of special primitive expressions is shown in Table 6.3, and is used to match boundaries of strings.

Syntax	What it represents
^	match an empty string at the beginning of a line
$	match an empty string at the end of a line
\b	match an empty string at a word boundary
\B	match an empty string not at a word boundary
\<	match an empty string at the beginning of a word
\>	match an empty string at the end of a word

Table 6.3: Primitives to match string boundaries

```
'hello world' matchesRegex: '.*\bw.*'    ⟶    true     "word boundary before w"
'hello world' matchesRegex: '.*\bo.*'    ⟶    false    "no boundary before o"
```

6.3 Regex API

Up to now we have focussed mainly on the syntax of regexes. Now we will have a closer look at the different messages understood by strings and regexes.

Matching prefixes and ignoring case

So far most of our examples have used the String extension method matchesRegex:.

Strings also understand the following messages: prefixMatchesRegex:, matchesRegexIgnoringCase: and prefixMatchesRegexIgnoringCase:.

The message prefixMatchesRegex: is just like matchesRegex, except that the whole receiver is not expected to match the regular expression passed as the argument; matching just a prefix of it is enough.

```
'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
```

Enumeration interface

Some applications need to access *all* matches of a certain regular expression within a string. The matches are accessible using a protocol modeled after the familiar Collection-like enumeration protocol.

regex:matchesDo: evaluates a one-argument aBlock for every match of the regular expression within the receiver string.

```
list := OrderedCollection new.
'Jack meet Jill' regex: '\w+' matchesDo: [:word | list add: word].
list  ⟶  an OrderedCollection('Jack' 'meet' 'Jill')
```

regex:matchesCollect: evaluates a one-argument aBlock for every match of the regular expression within the receiver string. It then collects the results and answers them as a SequenceableCollection.

```
'Jack meet Jill' regex: '\w+' matchesCollect: [:word | word size]  ⟶
    an OrderedCollection(4 4 4)
```

allRegexMatches: returns a collection of all matches (substrings of the receiver string) of the regular expression.

```
'Jack and Jill went up the hill' allRegexMatches: '\w+'        ⟶
    an OrderedCollection('Jack' 'and' 'Jill' 'went' 'up' 'the' 'hill')
```

Replacement and translation

It is possible to replace all matches of a regular expression with a certain string using the message copyWithRegex:matchesReplacedWith:.

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

A more general substitution is match translation. This message evaluates a block passing it each match of the regular expression in the receiver string and answers a copy of the receiver with the block results spliced into it in place of the respective matches.

```
'Krazy loves Ignatz' copyWithRegex: '\b[a−z]+\b' matchesTranslatedUsing: [:each | each
    asUppercase]   ⟶   'Krazy LOVES Ignatz'
```

All messages of enumeration and replacement protocols perform a case-sensitive match. Case-insensitive versions are not provided as part of a String protocol. Instead, they are accessible using the lower-level matching interface presented in the following question.

Lower-level interface

When you send the message matchesRegex: to a string, the following happens:

1. A fresh instance of RxParser is created, and the regular expression string is passed to it, yielding the expression's syntax tree.
2. The syntax tree is passed as an initialization parameter to an instance of RxMatcher. The instance sets up some data structure that will work as a recognizer for the regular expression described by the tree.
3. The original string is passed to the matcher, and the matcher checks for a match.

The Matcher

If you repeatedly match a number of strings against the same regular expression using one of the messages defined in String, the regular expression string is parsed and a new matcher is created for every match. You can avoid this

overhead by building a matcher for the regular expression, and then reusing the matcher over and over again. You can, for example, create a matcher at a class or instance initialization stage, and store it in a variable for future use. You can create a matcher using one of the following methods:

- You can send asRegex or asRegexIgnoringCase to the string.
- You can directly instantiate a RxMatcher using one of its class methods: forString: or forString:ignoreCase: (which is what the convenience methods above will do).

Here we send matchesIn: to collect all the matches found in a string:

```
octal := '8r[0−9A−F]+' asRegex.
octal matchesIn: '8r52 = 16r2A'    ⟶    an OrderedCollection('8r52')

hex := '16r[0−9A−F]+' asRegexIgnoringCase.
hex matchesIn: '8r52 = 16r2A'    ⟶    an OrderedCollection('16r2A')

hex := RxMatcher forString: '16r[0−9A−Fa−f]+' ignoreCase: true.
hex matchesIn: '8r52 = 16r2A'    ⟶    an OrderedCollection('16r2A')
```

Matching

A matcher understands these messages (all of them return true to indicate successful match or search, and false otherwise):

matches: aString – true if the whole argument string (aString) matches.

```
'\w+' asRegex matches: 'Krazy'    ⟶    true
```

matchesPrefix: aString – true if some prefix of the argument string (not necessarily the whole string) matches.

```
'\w+' asRegex matchesPrefix: 'Ignatz hates Krazy'    ⟶    true
```

search: aString – Search the string for the first occurrence of a matching substring. (Note that the first two methods only try matching from the very beginning of the string). Using the above example with a matcher for a+, this method would answer success given a string 'baaa', while the previous two would fail.

```
'\b[a−z]+\b' asRegex search: 'Ignatz hates Krazy'    ⟶    true    "finds 'hates'"
```

The matcher also stores the outcome of the last match attempt and can report it: lastResult answers a Boolean: the outcome of the most recent match attempt. If no matches were attempted, the answer is unspecified.

```
number := '\d+' asRegex.
number search: 'Ignatz throws 5 bricks'.
number lastResult    →    true
```

matchesStream:, matchesStreamPrefix: and searchStream: are analogous to the above three messages, but takes streams as their argument.

```
ignatz := ReadStream on: 'Ignatz throws bricks at Krazy'.
names := '\<[A−Z][a−z]+\>' asRegex.
names matchesStreamPrefix: ignatz    →    true
```

Subexpression matches

After a successful match attempt, you can query which part of the original string has matched which part of the regex. A subexpression is a parenthesized part of a regular expression, or the whole expression. When a regular expression is compiled, its subexpressions are assigned indices starting from 1, depth-first, left-to-right.

For example, the regex ((\\d+)\\s*(\\w+)) has four subexpressions, including itself.

```
1:  ((\d+)\s*(\w+))   "the complete expression"
2:  (\d+)\s*(\w+)     "top parenthesized subexpression"
3:  \d+               "first leaf subexpression"
4:  \w+               "second leaf subexpression"
```

The highest valid index is equal to 1 plus the number of matching parentheses. (So, 1 is always a valid index, even if there are no parenthesized subexpressions.)

After a successful match, the matcher can report what part of the original string matched what subexpression. It understands these messages:

subexpressionCount answers the total number of subexpressions: the highest value that can be used as a subexpression index with this matcher. This value is available immediately after initialization and never changes.

subexpression: takes a valid index as its argument, and may be sent only after a successful match attempt. The method answers a substring of the original string the corresponding subexpression has matched to.

subBeginning: and subEnd: answer the positions within the argument string or stream where the given subexpression match has started and ended, respectively.

```
items := '((\d+)\s*(\w+))' asRegex.
items search: 'Ignatz throws 1 brick at Krazy'.
```

```
items subexpressionCount    ⟶ 4
items subexpression: 1      ⟶ '1 brick'  "complete expression"
items subexpression: 2      ⟶ '1 brick'  "top subexpression"
items subexpression: 3      ⟶ '1'        "first leaf subexpression"
items subexpression: 4      ⟶ 'brick'    "second leaf subexpression"
items subBeginning: 3       ⟶ an OrderedCollection(14)
items subEnd: 3             ⟶ an OrderedCollection(15)
items subBeginning: 4       ⟶ an OrderedCollection(16)
items subEnd: 4             ⟶ an OrderedCollection(21)
```

As a more elaborate example, the following piece of code uses a MMM DD, YYYY date format recognizer to convert a date to a three-element array with year, month, and day strings:

```
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')
```

Enumeration and Replacement

The String enumeration and replacement protocols that we saw earlier in this section are actually implemented by the matcher. RxMatcher implements the following methods for iterating over matches within strings: matchesIn:, matchesIn:do:, matchesIn:collect:, copy:replacingMatchesWith: and copy:translatingMatchesUsing:.

```
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'
```

There are also the following methods for iterating over matches within streams: matchesOnStream:, matchesOnStream:do:, matchesOnStream:collect:, copyStream:to:replacingMatchesWith: and copyStream:to:translatingMatchesUsing:.

```
in := ReadStream on: '12 drummers, 11 pipers, 10 lords, 9 ladies, etc.'.
```

```
out := WriteStream on: ''.
numMatch := '\<\d+\>' asRegex.
numMatch
  copyStream: in
  to: out
  translatingMatchesUsing: [:each | each asNumber asFloat asString ].
out close; contents    ⟶    '12.0 drummers, 11.0 pipers, 10.0 lords, 9.0 ladies, etc.'
```

Error Handling

Several exceptions may be raised by RxParser when building regexes. The exceptions have the common parent RegexError. You may use the usual Smalltalk exception handling mechanism to catch and handle them.

- RegexSyntaxError is raised if a syntax error is detected while parsing a regex
- RegexCompilationError is raised if an error is detected while building a matcher
- RegexMatchingError is raised if an error occurs while matching (for example, if a bad selector was specified using ':<selector>:' syntax, or because of the matcher's internal error)

```
['+' asRegex] on: RegexError do: [:ex | ^ ex printString ]    ⟶
      'RegexSyntaxError:  nullable closure'
```

6.4 Implementation notes by Vassili Bykov

What to look at first. In 90% of the cases, the method String»matchesRegex: is all you need to access the package.

RxParser accepts a string or a stream of characters with a regular expression, and produces a syntax tree corresponding to the expression. The tree is made of instances of Rxs* classes.

RxMatcher accepts a syntax tree of a regular expression built by the parser and compiles it into a matcher: a structure made of instances of Rxm* classes. The RxMatcher instance can test whether a string or a positionable stream of characters matches the original regular expression, or it can search a string or a stream for substrings matching the expression. After a match is found, the matcher can report a specific string that matched the whole expression, or any parenthesized subexpression of it. All other classes support the same functionality and are used by RxParser, RxMatcher, or both.

Caveats. The matcher is similar in spirit, but *not* in design to Henry Spencer's original regular expression implementation in C. The focus is on simplicity, not on efficiency. I didn't optimize or profile anything. The matcher passes H. Spencer's test suite (see "test suite" protocol), with quite a few extra tests added, so chances are good there are not too many bugs. But watch out anyway.

Acknowledgments. Since the first release of the matcher, thanks to the input from several fellow Smalltalkers, I became convinced a native Smalltalk regular expression matcher was worth the effort to keep it alive. For the advice and encouragement that made this release possible, I want to thank: Felix Hack, Eliot Miranda, Robb Shecter, David N. Smith, Francis Wolinski and anyone whom I haven't yet met or heard from, but who agrees this has not been a complete waste of time.

6.5 Chapter summary

Regular expressions are an essential tool for manipulating strings in a trivial way. This chapter presented the Regex package for Pharo. The essential points of this chapter are:

- For simple matching, just send `matchesRegex:` to a string
- When performance matters, send `asRegex` to the string representing the regex, and reuse the resulting matcher for multiple matches
- Subexpression of a matching regex may be easily retrieved to an arbitrary depth
- A matching regex can also replace or translate subexpressions in a new copy of the string matched
- An enumeration interface is provided to access all matches of a certain regular expression
- Regexes work with streams as well as with strings.