Regular Expressions

A little theory

Good news and Bad news

The match operator

Character classes

Alternation and grouping

Repetition

split and scan

Anchors

Grouping and references

Iteration with gsub

Application: Time totaling

A little theory

In computer science theory, a language is a set of strings. The set may be infinite.

The Chomsky hierarchy of languages looks like this:

Unrestricted languages	("Type 0")
Context-sensitive languages	("Type 1")
Context-free languages	("Type 2")
Regular languages	("Type 3")

Roughly speaking, natural languages are *unrestricted languages* that can only specified by *unrestricted grammars*.

Programming languages are usually *context-free languages*—they can be specified with a *context-free grammar*, which has very restrictive rules. Every Java program is a string in the context-free language that is specified by the Java grammar.

A *regular language* is a very limited kind of context free language that can be described by a *regular grammar*. <u>A regular language can also be described by a *regular expression*.</u>

A little theory, continued

A regular expression is simply a string that may contain metacharacters. Here is a simple regular expression:

```
a+
```

It specifies the regular language that consists of the strings {a, aa, aaa, ...}.

Here is another regular expression:

```
(ab)+c*
```

It describes the set of strings that have ab repeated some number of times followed by zero or more c's. Some strings in the language are ab, ababc, and abababababccccccc.

The regular expression

```
(north|south)(east|west)
```

describes a language with four strings: {northeast, northwest, southeast, southwest}.

Good news and bad news

UNIX tools such as the ed editor and grep/fgrep/egrep introduced regular expressions to a wide audience.

Many languages provide a library for working with regular expressions. Java provides the java.util.regex package. The command man regex produces some documentation for the C library's regular expression routines.

Some languages, Ruby included, have a regular expression datatype.

Regular expressions have a sound theoretical basis and are also very practical. Over time, however, a great number of extensions have been added. In languages like Ruby, regular expressions are truly a language within a language.

Chapter 22 of the text devotes four pages to its summary of regular expressions. In contrast, integers, floating point numbers, strings, ranges, arrays, and hashes are summarized in a total of four pages.

Good news and Bad news, continued

Entire books have been written on the subject of regular expressions. A number of tools have been developed to help programmers create and maintain complex regular expressions.

Here is a regular expression written by Mark Cranness and posted at regexlib.com:

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?=[\x01-\x7f])[^"\\]|\\[\x01-\x7f])*"\x20*)*(?
<angle><))?((?!\.)(?>\.?[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+)+|"((?=[\x01-\x7f])[^"\\]|\\[\x01-\
x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]{2,}|\[(((?(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?
\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-\x7f])[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

It describes RFC 2822 email addresses.

The instructor believes that regular expressions have their place but grammar-based parsers should be considered more often than they are, especially when an underlying specification includes a grammar.

We'll cover a subset of Ruby's regular expression capabilities.

A simple regular expression in Ruby

One way to create a regular expression (RE) in Ruby is to use the /*pattern*/ syntax:

```
>> re = /a.b.c/        => /a.b.c/

>> re.class            => Regexp
```

In an RE, a dot is a *metacharacter* (a character with special meaning) that will match any (one) character.

Alphanumeric characters and some special characters simply match themselves.

The meaning of a metacharacter can be suppressed by preceding with a backslash.

The RE /a.b.c/ matches strings that contain the five-character sequence a<*anychar*>b<*anychar*>c, like "albacore", "barbecue", "drawback", and "iambic".

How many strings are in the language specified with the regular expression /a.b.c/?

The match operator

The binary operator =~ is called "match". One operand must be a string and the other must be a regular expression. If the string contains a match for the RE, the position of the match is returned. nil is returned if there is no match.

```
>> "albacore" =~ /a.b.c/            => 0

>> /a.b.c/ =~ "drawback"            => 2

>> "abc" =~ /a.b.c/                 => nil
```

What does the following loop do?

```
while line = gets do
   puts line if line =~ /a.b.c/
end
```

How could we invert the operation of the loop?

Problem: Write a program that prints lines longer than the length specified by a command line argument. For example, longerthan 80 < x prints the lines in x that are 81 characters or more in length. (Don't use String#length or size!)

The match operator, continued

After a successful match we can use some cryptically named predefined variables to access parts of the string:

$`	Is the portion of the string that precedes the match. (That's a backquote.)
$&	Is the portion of the string that was matched by the regular expression.
$'	Is the portion of the string following the match.

Example:

```
>> "limit=300" =~ /=/          => 5

>> $`                          => "limit"

>> $&                          => "="

>> $'                          => "300"
```

The match operator, continued

Here is a handy utility routine from the text:

```
def show_match(s, re)
    if s =~ re then
        "#{$`}<<#{$&}>>#{$'}"
    else
        "no match"
    end
end
```

Usage:

```
>> show_match("limit is 300", /is/)            => "limit <<is>> 300"

>> %w{albacore drawback iambic}.each { |w| puts show_match(w, /a.b.c/) }
<<albac>>ore
dr<<awbac>>k
i<<ambic>>
```

Handy: Put show_match in your ~/.irbrc file. Maybe name it sm.

Regular expressions as subscripts

As a subscript, a regular expression specifies the portion of the string, if any, matched by it.

```
>> s = "testing"            => "testing"

>> s[/.../] = "*"           => "*"

>> s                        => "*ting"
```

Another example:

```
>> %w{albacore drawback iambic}.map { |w| w[/a.b.c/] = "(a.b.c)"; w }
=> ["(a.b.c)ore", "dr(a.b.c)k", "i(a.b.c)"]
```

Character classes

The pattern [*characters*] is an RE that matches any one of the specified *characters*.

[^characters] is an RE that matches any character not in the set. (It matches the *complement* of the set.)

A dash between two characters in a set specifies a range based on ASCII codes.

Examples:

/[aeiou]/ matches a string that contains a lower-case vowel

```
>> show_match("testing", /[aeiou]/)
=> "t<<e>>sting"
```

/[^0-9]/ matches a string that contains a non-digit

```
>> show_match("1,000", /[^0-9]/)
=> "1<<,>>000"
```

/[a-z][0-9][a-z]/ matches strings that somewhere contain the three-character sequence lowercase letter, digit, lowercase letter.

```
>> show_match("A1b33s4ax1", /[a-z][0-9][a-z]/)
=> "A1b33<<s4a>>x1"
```

Character classes, continued

Ruby provides abbreviations for some commonly used character classes:

\d	Stands for [0-9]
\w	Stands for [A-Za-z0-9_]
\s	Whitespace—blank, tab, carriage return, newline, formfeed

The abbreviations \D, \W, and \S produce a complemented set for the corresponding class.

Examples:

```
>> show_match("Call me at 555-1212", /\d\d\d-\d\d\d\d/)
=> "Call me at <<555-1212>>"

>> "fun double(n) = n * 2".gsub(/\w/,".")
=> "... ......(.) = . * ."

>> "FCS 202, 12:30-13:45 TH".gsub(/\D/, "~")
=> "~~~~202~~12~30~13~45~~~"
```

gsub's replacement string can be any length, as you'd expect.

Alternatives and grouping

Alternatives can be specified with a vertical bar:

```
>> %w{one two three four}.select { |s| s =~ /two|four|six/ }
=> ["two", "four"]
```

Parentheses can be used for grouping. Consider this regular expression:

```
/(two|three) (apple|biscuit)s/
```

It corresponds to a regular language with four strings:

```
two apples
three apples
two biscuits
three biscuits
```

Usage:

```
>> "I ate two apples." =~ /(two|three) (apple|biscuit)s/      => 6

>> "She ate three mice." =~ /(two|three) (apple|biscuit)s/ => nil
```

Creating regular expressions at run-time

The method Regexp.new(s) creates a regular expression from the string s.

```
counts = %w{two three four five}
foods = %w{apples oranges bananas}

re = ""; sep = ""
counts.each {
    |count| foods.each {
        |food|
        re << sep << count << " " << food
        sep = "|"
    }
}
puts re
re = Regexp.new(re)
while line = (printf("Query? "); gets)
    if line =~ re then
        puts "Yes: #{$`}[#{$&}]#{$'}"
    else puts "No"
    end
end
```

Execution:

% **ruby re2.rb**
two apples|two oranges|two bananas|three apples|three oranges|three bananas|four apples|...

Query? **Are there four apples?**
Yes: Are there [four apples]?

Query? **We sold two bananas.**
Yes: We sold [two bananas].

Query? **Three oranges were thrown at me!**
No

Repetition with *, +, and ?

If R is a regular expression, then...

*R** matches <u>zero or more</u> occurrences of *R*.

R+ matches <u>one or more</u> occurrences of *R.*

R? matches <u>zero or one</u> occurrences of *R.*

All have higher precedence than juxtaposition.

Examples:

/ab*c/ Matches strings that contain an 'a' that is followed by <u>zero or more</u> 'b's that are followed by a 'c'. Examples: ac, abc, abbbbbbc, back, and cache.

/-?\d+/ Matches strings that contain an integer. What strings are matched by /-?\d*/? What would show_match("maybe --123.4e-10 works", /-?\d+/) produce?

/a(12|21|3)*b/
Matches strings like ab, a3b, a312b, and a3123213123333b.

Repetition, continued

The operators *, +, and ? are "greedy"—each tries to match the longest string possible, and cuts back only to make the full expression succeed. Example:

Given a.*b and the input 'abbb', the first attempt is:

a	matches a
.*	matches bbb
b	*fails*—no characters left!

The matching algorithm then *backtracks* and does this:

a	matches a
.*	matches bb
b	matches b

Repetition, continued

More examples of greed:

```
>> show_match("xabbbbc", /a.*b/)          => "x<<abbbb>>c"

>> show_match("xabbbbc", /ab?b?/)         => "x<<abb>>bbc"

>> show_match("xabbbbc", /ab?b?.*c/       => "x<<abbbbc>>"

>> show_match("maybe --123.4e-10 works", /-?\d+/)
=> "maybe -<<-123>>.4e-10 works"
```

Why are *, +, and ? greedy?

Repetition, continued

Describe the strings matched by...

```
/[a-z]+[0-9]?/
/a...b?c/
/..1.*2../
/..*.+.*./
/((ab)+c?(xyz)*)?/
```

Specify an RE that matches...

Strings corresponding to ML int lists, like [10], [5,1,~700], and [ ]. Assume there are no embedded spaces.

Lines that contain only whitespace and a left or right brace.

Strings that match /^[A-Za-z_]\w*$/ commonly occur in programs. What are they?

split and scan with regular expressions

It is possible to split a string on a regular expression:

```
>> "  one, two,three / four".split(/[\s,\/]+/)    # Note escaped backslash in class
=> ["", "one", "two", "three", "four"]
```

Unfortunately, leading delimiters produce an empty string in the result.

If we can describe the strings of interest instead of what separates them, scan is a better choice:

```
>> "10.0/-1.3...5.700+[1.0,2.3]".scan(/-?\d+\.\d+/)
=> ["10.0", "-1.3", "5.700", "1.0", "2.3"]
```

Here's a way to keep all the pieces:

```
>> "  one, two,three / four".scan(/(\w+|\W+)/)
=> [["  "], ["one"], [", "], ["two"], [","], ["three"], [" / "], ["four"]]
```

A list of lists is produced because of the grouping. We'll see a use for this later.

Anchors

The metacharacter ^ is an *anchor*. It doesn't match any characters but it constrains the following regular expression to appear at the beginning of the string being matched against.

Another anchor is $. It constrains the preceding regular expression to appear at the end of the string.

```
$ grep.rb  ^bucket  <  $words
bucket
bucketed
bucketeer

$ grep.rb  bucket$  <  $words
bucket
gutbucket
trebucket
```

Problems:

Specify an RE that will match words that are at least six characters long, start with an 'a', and end with a 'z'.

Count the number of empty lines in x.rb. (Yes, you can't use String#size!)

Groups and references

In addition to providing a way to override precedence rules, parentheses create *references* (also called *back references*) to the text matched by a group.

Here is a regular expression that matches strings consisting of digits where the first and last digit are the same:

```
/^(\d)\d*\1$/
```

Piece by piece:

^	Require the following RE to be at the beginning of the string.
(\d)	Match one digit and retain it as the text of "group 1".
\d*	Match zero or more digits.
\1	The text of group 1.
$	Require the preceding RE to be at the end of the string.

Groups and references, continued

For reference:

```
/^(\d)\d*\1$/
```

Usage:

```
>> show_match("121", /^(\d)\d*\1$/)          => "<<121>>"

>> show_match("12", /^(\d)\d*\1$/)           => "no match"

>> show_match("3013", /^(\d)\d*\1$/)         => "<<3013>>"

>> show_match("3", /^(\d)\d*\1$/)            => "no match"
```

A little fun:

```
>> (1000..2000).select { |n| (7**n).to_s =~ /^(\d)\d*\1$/ }
=> [1000, 1012, 1020, 1021, 1023, 1032, 1044, 1046, 1052, 1053, 1055, 1064, 1075,
1084, 1096, 1107, 1116, 1128, 1130, 1136, 1137, 1139, 1148, 1168, 1180, 1191,
1200, 1212, 1220, 1221, 1223, 1232, 1246, 1252, 1255, 1264, 1275, 1284, ... ]
```

Groups and references, continued

In addition to setting $`, $&, and $', a successful match also sets $1, $2, ..., $9 to the text of the corresponding group.

Strictly to illustrate the mechanism, here is a method that swaps the first three and last characters of a string:

```
def swap3(s)
    if s =~ /(...)(.*)(...)/ then
        "#{$3}#{$2}#{$1}"
    else
        s
    end
end
```

Usage:

```
>> swap3 "abc-def"          => "def-abc"
>> swap3 "aaabbb"           => "bbbaaa"
>> swap3 "abcd"             => "abcd"
```

In actual practice what's a better way to perform this computation?

Groups and references, continued

As a more practical example, here is a method that rewrites infix operators as function calls:

```
$ops = { "-" => "sub", "+" => "add", "mul" => "mul", "div" => "div" }  # global variable
def infix_to_function(line)
    if line =~ /^(\w+)\s*(([-+]|(mul|div)))\s*(\w+)$/ then
        fcn = $ops[$2]
        return "#{fcn}(#{$1},#{$5})"
    else
        return nil
    end
end
```

Usage:

```
>> infix_to_function("3 + 4")              => "add(3,4)"

>> infix_to_function("limit-1500")         => "sub(limit,1500)"

>> infix_to_function("10mul20")            => "mul(10,20)"
```

Could we generate the regular expression from the hash? Do we really need a character class or would just alternation suffice?

Groups and references, continued

If the argument to scan has one or more groups, a list of lists is produced:

```
>> "1:2  33:28  100:7".scan(/(\d+):(\d+)/)
=> [["1", "2"], ["33", "28"], ["100", "7"]]

>> "1234567890".scan(/(.)(.)(.)/)
=> [["1", "2", "3"], ["4", "5", "6"], ["7", "8", "9"]]
```

Recall this example:

```
>> "  one, two,three / four".scan(/(\w+|\W+)/)
=> [["  "], ["one"], [", "], ["two"], [","], ["three"], [" / "], ["four"]]
```

Iteration with gsub

Recall String#gsub:

```
>> "fun double(n) = n * 2".gsub(/\w/,".")
=> "... ......(.) = . * ."
```

gsub has a one argument form that is an iterator. The result of the block is substituted for the match.

Here is a method that augments a string with a running sum of the numbers it contains:

```
def running_sums(s)
    sum = 0
    s.gsub(/\d+/) {
        sum += $&.to_i
        $& + "(%d)" % sum
        }
end
```

Usage:

```
>> running_sum("1 pencil, 3 erasers, 2 pens")
=> "1(1) pencil, 3(4) erasers, 2(6) pens"
```

Application: Time totaling

Consider an application that reads elapsed times on standard input and prints their total:

```
% ttl.rb
3h
15m
4:30
^D
7:45
```

Multiple times can be specified per line:

```
% ruby ttl.rb
10m, 20m
3:30 2:15 1:01
^D
7:16
```

Times in an unexpected format are ignored:

```
% ttl.rb
10 2:90
What's 10?  Ignored...
What's 2:90?  Ignored...
```

Time totaling, continued

```
def main
   mins = 0
   while line = gets do
      line.scan(/[^\s,]+/).each {|time| mins += parse_time(time) }
   end
   printf("%d:%02d\n", mins / 60, mins % 60)
end

def parse_time(s)
   case
   when s =~ /^(\d+):([0-5]\d)$/
      $1.to_i * 60 + $2.to_i
   when s =~ /^(\d+)([hm])$/
      if $2 == "h" then  $1.to_i * 60
                    else $1.to_i end
   else
      print("What's #{s}?  Ignored...\n");  0
   end
end
main
```