Review of Basic Perl and Perl Regular Expressions

Alexander Fraser & Liane Guillou
{fraser,liane}@cis.uni-muenchen.de

CIS, Ludwig-Maximilians-Universität München

Computational Morphology and Electronic Dictionaries
SoSe 2016
2016-05-02

Outline

- Today will start with a review of Perl
- Followed by Perl regular expressions
  – Regular expressions are closely tied to the Finite State Acceptors (and Transducers) we saw last time

Credits

Adapted from Perl Tutorial - Bioinformatics Orientation 2008 By Eric Bishop which was: Adapted from slides found at: www.csd.uoc.gr/~hy439/Perl.ppt (original author is not indicated)

Why Perl?

- Perl is built around regular expressions
  - REs are good for string processing
  - Therefore Perl is a good scripting language
  - Perl is especially popular for CGI scripts
- Perl makes full use of the power of UNIX
- Short Perl programs can be very short
  - "Perl is designed to make the easy jobs easy, without making the difficult jobs impossible." -- Larry Wall, *Programming Perl*

Why not Perl?

- Perl is *very* UNIX-oriented
  - Perl is available on other platforms...
  - ...but isn't always fully implemented there
  - However, Perl is often the best way to get some UNIX capabilities on less capable platforms
- Perl does not scale well to large programs
  - Weak subroutines, heavy use of global variables
- Perl's syntax is not particularly appealing

Perl Example 1

```
#!/usr/bin/perl -w
#
# Program to do the obvious
#
print 'Hello world.';     # Print a message
```

Understanding "Hello World"

- Comments are # to end of line
  - But the first line, #!/usr/bin/perl, tells where to find the Perl compiler on your system
  - I use the modifier "-w" to get extra warnings, highly recommended
- Perl statements end with semicolons
- Perl is case-sensitive

Running your program

- Two ways to run your program:
  - perl hello.pl
  - chmod 700 hello.pl
    ./hello.pl

Scalar variables

- Scalar variables start with $
- Scalar variables hold strings or numbers, and they are interchangeable
- When you first use (declare) a variable use the **my** keyword to indicate the variable's scope
  - Without "use strict;", this is not necessary but good programming practice
  - With "use strict;", won't compile (highly recommended!)

- Example:
  - use strict;
  - my $priority = 9;

Arithmetic in Perl

```
$a = 1 + 2;      # Add 1 and 2 and store in $a
$a = 3 - 4;      # Subtract 4 from 3 and store in $a
$a = 5 * 6;      # Multiply 5 and 6
$a = 7 / 8;      # Divide 7 by 8 to give 0.875
$a = 9 ** 10;    # Nine to the power of 10, that is, 9^10
$a = 5 % 2;      # Remainder of 5 divided by 2
++$a;            # Increment $a and then return it
$a++;            # Return $a and then increment it
--$a;            # Decrement $a and then return it
$a--;            # Return $a and then decrement it
```

Arithmetic in Perl cont'd

- You sometimes may need to group terms
  - Use parentheses ()
  - (5-6)*2 is not 5-(6*2)

String and assignment operators

```
$a = $b . $c;   # Concatenate $b and $c
$a = $b x $c;   # $b repeated $c times

$a = $b;        # Assign $b to $a
$a += $b;       # Add $b to $a
$a -= $b;       # Subtract $b from $a
$a .= $b;       # Append $b onto $a
```

Single and double quotes

- $a = 'apples';
- $b = 'bananas';
- print $a . ' and ' . $b;
  - prints: apples and bananas
- print '$a and $b';
  - prints: $a and $b
- print "$a and $b";
  - prints: apples and bananas

Perl Example 2

```
#!/usr/bin/perl -w
# program to add two numbers

use strict;

my $a = 3;
my $b = 5;
my $c = “the sum of $a and $b and 9 is: ”;
my $d = $a + $b + 9;
print “$c $d\n”;
```

if statements

```
if ($a eq “”)
{
	print "The string is empty\n";
}
else
{
	print "The string is not empty\n";
}
```

Tests

- All of the following are *false*:
  0, '0', "0", '', "", "Zero"
- Anything not *false* is *true*
- Use == and != for numbers, eq and ne for strings
- &&, ||, and ! are *and*, *or*, and *not*, respectively.

if - elsif statements

```
if ($a eq “”)
  { print "The string is empty\n"; }
elsif (length($a) == 1)
  { print "The string has one character\n"; }
elsif (length($a) == 2)
  { print "The string has two characters\n"; }
else
  { print "The string has many characters\n"; }
```

while loops

```
#!/usr/bin/perl –w

use strict;

my $i = 5;
while ($i < 15)
{
    print ”$i";
    $i++;
}
```

for loops

- ```
  for (my $i = 5; $i < 15; $i++)
  {
          print "$i\n";
  }
  ```
  ```

last

- The **last** statement can be used to exit a loop before it would otherwise end

```
for (my $i = 5; $i < 15; $i++)
{
        print "$i,";
        if($i == 10)
        {
                last;
        }
}
print “\n”;
```

when run, this prints 5,6,7,8,9,10

next

- The **next** statement can be used to end the current loop iteration early

```
for (my $i = 5; $i < 15; $i++)
{
        if($i == 10)
        {
                next;
        }
         print "$i,";
}
print “\n”
```

when run, this prints 5,6,7,8,9,11,12,13,14

Standard I/O

- On the UNIX command line;
  - **< *filename*** means to get input from this file
  - **> *filename*** means to send output to this file
- STDIN is standard input
  - To read a line from standard input use:

```
my $line = <STDIN>;
```

- STDOUT is standard output
  - Print will output to STDOUT by default
  - You can also use :

```
print STDOUT "my output goes here";
```

File I/O

- Often we want to read/write from specific files
- In perl, we use **file handles** to manipulate files
- The syntax to open a handle to read to a file for **reading** is different than opening a handle for **writing**
  - To open a file handle for reading:

    open IN, “<fileName”;
  - To open a file handle for writing:

    open OUT, “>fileName”;

- File handles must be closed when we are finished with them -- this syntax is the same for all file handles

    close IN;

File I/O cont’d

- Once a file handle is open, you may use it just like you would use STDIN or STDOUT
- To read from an open file handle:
  - my $line = <IN>;
- To write to an open file handle:
  - print OUT “my output data\n”;

Perl Example 3

```
#!/usr/bin/perl -w
# singlespace.pl: remove blank lines from a file
# Usage: perl singlespace.pl < oldfile > newfile
use strict;
while (my $line = <STDIN>)
{
    if ($line eq "\n")
    {
        next;
    }
    print "$line";
}
```

Arrays

- my @food = ("apples", "bananas", "cherries");
- But…
- print $food[1];
  - prints "bananas"
- my @morefood = ("meat", @food);
  - @morefood now contains:
    ("meat", "apples", "bananas", "cherries");

push and pop

- push adds one or more things to the end of a list
  - push (@food, "eggs", "bread");
  - push returns the new length of the list
- pop removes and returns the last element
  - $sandwich = pop(@food);
- $len = @food; # $len gets length of @food
- $#food # returns index of last element

@ARGV: a special array

- A special array, @ARGV, contains the parameters you pass to a program on the command line

- If you run “perl test.pl a b c”, then within test.pl @ARGV will contain (“a”, “b”, “c”)

foreach

```
# Visit each item in turn and call it $morsel

foreach my $morsel (@food)
{
	print "$morsel\n";
	print "Yum yum\n";
}
```

Hashes / Associative arrays

- Associative arrays allow lookup by name rather than by index
- Associative array names begin with %
- Example:
  - my %fruit = ("apples”=>"red", "bananas”=>"yellow", "cherries”=>"red");

  - Now, $fruit{"bananas"} returns "yellow”

  - To set value of a hash element:
    $fruit{“bananas”} = “green”;

Hashes / Associative Arrays II

- To remove a hash element use **delete**
  - delete $fruit{“bananas”};

- You cannot index an associative array, but you can use the **keys** and **values** functions:

```
foreach my $f (keys %fruit)
{
   print ("The color of $f is " . $fruit{$f} . "\n");
}
```

Example 4

```
#!/usr/bin/perl -w

use strict;

my @names = ( "bob", "sara", "joe" );
my %likesHash = ( "bob"=>"steak", "sara"=>"chocolate",
    "joe"=>"rasberries" );

foreach my $name (@names)
{
    my $nextLike = $likesHash{$name};
    print "$name likes $nextLike\n";
}
```

Regular Expressions

- $sentence =~ /the/
  - True if $sentence contains "the"
- $sentence = "The dog bites.";
  if ($sentence =~ /the/) # is *false*
  - …because Perl is case-sensitive
- !~ is "does not contain"

RE special characters

.   # Any single character except a newline

^   # The beginning of the line or string

$ # The end of the line or string

*   # Zero or more of the last character

+   # One or more of the last character

?   # Zero or one of the last character

RE examples

```
^.*$        # matches the entire string
hi.*bye     # matches from "hi" to "bye" inclusive
x +y        # matches x, one or more blanks, and y
^Dear       # matches "Dear" only at beginning
bags?       # matches "bag" or "bags"
hiss+       # matches "hiss", "hisss", "hissss", etc.
```

Square brackets

```
[qjk]        # Either q or j or k
[^qjk]       # Neither q nor j nor k
[a-z]        # Anything from a to z inclusive
[^a-z]       # No lower case letters
[a-zA-Z]     # Any letter
[a-z]+       # Any non-zero sequence of
             #   lower case letters
```

More examples

[aeiou]+ # matches one or more vowels

[^aeiou]+ # matches one or more nonvowels

[0-9]+ # matches an unsigned integer

[0-9A-F] # matches a single hex digit

[a-zA-Z] # matches any letter

[a-zA-Z0-9_]+ # matches identifiers

More special characters

\n # A newline
\t # A tab
\w # Any alphanumeric; same as [a-zA-Z0-9_]
\W # Any non-word char; same as [^a-zA-Z0-9_]
\d # Any digit. The same as [0-9]
\D # Any non-digit. The same as [^0-9]
\s # Any whitespace character
\S # Any non-whitespace character
\b # A word boundary, outside [] only
\B # No word boundary

Quoting special characters

```
\|      # Vertical bar
\[      # An open square bracket
\)      # A closing parenthesis
\*      # An asterisk
\^      # A carat symbol
\/      # A slash
\\      # A backslash
```

Alternatives and parentheses

```
jelly|cream   # Either jelly or cream

(eg|le)gs     # Either eggs or legs

(da)+         # Either da or dada or
              #    dadada or...
```

The $_ variable

- Often we want to process one string repeatedly
- The $_ variable holds the *current string*
- If a subject is omitted, $_ is assumed
- Hence, the following are equivalent:
  - if ($sentence =~ /under/) …
  - $_ = $sentence; if (/under/) …

Case-insensitive substitutions

- s/london/London/i
  - case-insensitive substitution; will replace london, LONDON, London, LoNDoN, etc.
- You can combine global substitution with case-insensitive substitution
  - s/london/London/gi

split

- split breaks a string into parts
- $info = "Caine:Michael:Actor:14, Leafy Drive";
  @personal = split(/:/, $info);
- @personal =
      ("Caine", "Michael", "Actor", "14, Leafy Drive");

Example 5

```
#!/usr/bin/perl –w
use strict;
my @lines = ( "Boston is cold.",
              "I like the Boston Red Sox.",
              "Boston drivers make me see red!" );
foreach my $line (@lines)
{
        if ($line =~ /Boston.*red/i )
        {
                print "$line\n";
        }
}
```

Calling subroutines

- Assume you have a subroutine printargs that just prints out its arguments
- Subroutine calls:
  - printargs("perly", "king");
    - Prints: "perly king"
  - printargs("frog", "and", "toad");
    - Prints: "frog and toad"

Defining subroutines

- Here's the definition of printargs:
  - sub printargs
    { print join(“ “, @_) . ”\n"; }
  - Parameters for subroutines are in an array called @_
  - The join() function is the opposite of split()
    - Joins the strings in an array together into one string
    - The string specified by first argument is put between the strings in the arrray

Returning a result

- The value of a subroutine is the value of the last expression that was evaluated

```
sub maximum
{
        if ($_[0] > $_[1])
          { $_[0]; }
        else
          { $_[1]; }
}

$biggest = maximum(37, 24);
```

Returning a result (cont'd)

- You can also use the "return" keyword to return a value from a subroutine
  - This is better programming practice

```
sub maximum
{
        my $max = $_[0];
        if ($_[1] > $_[0])
          { max = $_[1]; }
        return $max;
}
$biggest = maximum(37, 24);
```

Example 6

```
#!/usr/bin/perl -w
use strict;
sub inside
{
	my $a = shift @_;
	my $b = shift @_;
	$a =~ s/ //g;
	$b =~ s/ //g;
	return ($a =~ /$b/ || $b =~ /$a/);
}
if( inside("lemon", "dole money") )
{
		print "\"lemon\" is in \"dole money\"\n";
}
```

Engineering Regular Expressions

- There are some nice online packages and websites that can help with this.

- Let's look at a regular expression for recognizing simple floating point numbers like:
- 1
- -1
- -1.56
- +200000.5

- (Credit for basic idea to TCL manual, version 8.5)

- /[-+]?([0-9])*\.?([0-9]*)/

- Does this seem reasonable?

- We can go to regexper.com, and put in this regular expression and visualize it

Regexper

https://regexper.com/#%5B-%2B%5D%3F(%5B0-9%5D)*%5C.%3F(%5

REGEXPER

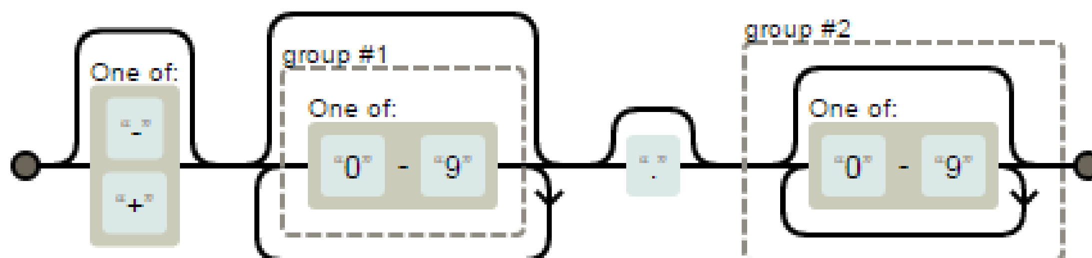
You thought you only had two problems...

Changelog
Documentation
Source on GitHub

```
[-+]?([0-9])*\.?([0-9]*)
```

Display

Download // Permalink



Created by Jeff Avallone // Generated images licensed: (cc) BY

- We can test our regular expression against strings at regex101.com

Online regex tester and de ✕

https://regex101.com

regex101

SAVE & S...

FLAVOR

PCRE

JS

PY

TOOLS

REGULAR EXPRESSION

1 MATCH - 11 STEPS

```
/ [-+]?([0-9])*\.?([0-9]*) / gmixXsuUAJ
```

TEST STRING

```
1.10
```

SUBSTITUTION

EXPLANATION

```
/ [-+]?([0-9])*\.?([0-9]*)
/
```

[-+]? **match a single character present in the list below**

Quantifier: ? Between zero

MATCH INFORMATION

MATCH 1

1.	[0-1]	`1`
2.	[2-4]	`10`

QUICK REFERENCE

FULL...

MOST USED TOKE...

most use...

A single cha... [abc]

all tokens

A characte... [^abc]

- Looks good, right?
- But... What is up with match 1 on the next slide?
- Credit here to Veronika Hintzen for noticing and explaining this bug in class!

Online regex tester and de

https://regex101.com

regex101

SAVE & S...

FLAVOR

PCRE

JS

PY

TOOLS

$

REGULAR EXPRESSION

1 MATCH - 17 STEPS

```
/ [-+]?([0-9])*\.?([0-9]*) / gmixXsuUAJ
```

TEST STRING

```
1000.1000
```

SUBSTITUTION

EXPLANATION

```
/ [-+]?([0-9])*\.?([0-9]*) /
```

[-+]? **match a single character present in the list below**

Quantifier: ? Between zero

MATCH INFORMATION

MATCH 1

1.	[3-4]	`0`
2.	[5-9]	`1000`

QUICK REFERENCE

FULL...

most use...

all tokens

MOST USED TOKE...

A single cha... [abc]

A characte... [^abc]

- Let's go back to the regexper.com graphic (back a few slides)
- Look at the first group. It looks different from the second group
- We can fix this by changing the regular expression to be like this (we move the first star inside the parenthesis):
- /[-+]?([0-9]*)\.?([0-9]*)/

- regex101.com allows us to test our new regular expression
  - Now it works as expected!

perlretut

- Final word: if you really want to master regular expressions, take a look at:
- perlretut
- The perl regular expressions tutorial

Thank you for your attention