COMP284 Scripting Languages

Lecture 6: Perl (Part 5)

Handouts (8 on 1)

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Substitutions: Capture variables

s/*regexpr*/*replacement*/

- Perl treats *replacement* like a double-quoted string

  ⇝ backslash escapes work as in a double-quoted string

\n	Newline
\t	Tab
\l	Lower case next letter
\L	Lower case all following letters until \E
\u	Upper case next letter
\U	Upper case all following letters until \E

  ⇝ variable interpolation is applied, including capture variables

$*N*	string matched by capture group *N* (where *N* is a natural number)
$+{*name*}	string matched by a named capture group

Contents

Substitutions: Capture variables

Example:

```
$name = "Dr␣Ullrich␣Hustadt";
$name =~ s/(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/\U$3\E, $2/;
print "$name\n";

$name = "Dave␣Shield";
$name =~ s/(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/\U$3\E, $2/;
print "$name\n";
```

Output:

```
HUSTADT, Ullrich
SHIELD, Dave
```


Substitutions

s/*regexpr*/*replacement*/

- Searches a variable for a match for *regexpr*, and if found, replaces that match with a string specified by *replacement*
- In both scalar context and list context returns the number of substitutions made (that is, 0 if no substitutions occur)
- If no variable is specified via one of the binding operators =~ or !~, the special variable $_ is searched and modified
- The binding operator !~ only negates the return value but does not affect the manipulation of the text

The delimiter / can be replaced by some other paired or non-paired character, for example:

s!*regexpr*!*replacement*! or s<*regexpr*>[*replacement*]

Substitutions: Modifiers

Modifiers for substitutions include the following:

s/ / /g	Match and replace globally, that is, all occurrences
s/ / /i	Case-insensitive pattern matching
s/ / /m	Treat string as multiple lines
s/ / /s	Treat string as single line
s/ / /e	Evaluate the right side as an expression

Combinations of these modifiers are also allowed

Example:

```
$_ = "Yabba␣dabba␣doo";
s/bb/dd/g;
print $_,"\n";
```

Output:

```
Yadda dadda doo
```


Substitutions

Example:

```
$text = "http://www.myorg.co.uk/info/refund/../vat.html";
$text =~ s!/[^\/]+/\.\.!!;
print "$text\n";
```

Output:

```
http://www.myorg.co.uk/info/vat.html
```

Example:

```
$_ = "Yabba␣dabba␣doo";
s/bb/dd/;
print $_,"\n";
```

Output:

```
Yadda dabba doo
```

Note: Only the first match is replaced

Substitutions: Modifiers

Modifiers for substitutions include the following:

s/ / /e	Evaluate the right side as an expression

Example:

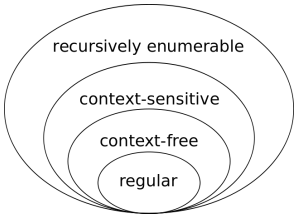
```
1 $text = "The␣temperature␣is␣105␣degrees␣Fahrenheit";
2 $text =~ s!(\d+) degrees Fahrenheit!
3            (($1-32)*5/9)."␣degrees␣Celsius"!e;
4 print "$text\n";
5 $text =~ s!(\d+\.\d+)!sprintf("%d",$1+0.5)!e;
6 print "$text\n";
```

Output:

```
The temperature is 40.5555555555556 degrees Celsius
The temperature is 41 degrees Celsius
```

Regular Expressions and the Chomsky Hierarchy

- In Computer Science, formal languages are categorised according to the type of grammar needed to generate them (or the type of automaton needed to recognise them)
- Perl regular expressions can at least recognise all context-free languages



Chomsky Hiearchy of Formal Languages

- However, this does not mean regular expression should be used for parsing context-free languages
- Instead there are packages specifically for parsing context-free languages or dealing with specific languages, e.g. HTML, CSV

Java methods versus Perl subroutines

- Java uses methods as a means to encapsulate sequences of instructions
- In Java you are expected
  - to declare the type of the return value of a method
  - to provide a list of parameters, each with a distinct name, and to declare the type of each parameter

```
public static int sum2( int f, int s) {
  f = f+s;
  return f;
}

public static void main(String[] args) {
  System.out.printf("sum of 3 and 4 is %d", sum2(3, 4));
}
```

- Instead of methods, Perl uses subroutines

Subroutines

Subroutines are defined as follows in Perl:

```
sub identifier {
  statements
}
```

- Subroutines can be placed anywhere in a Perl script but preferably they should all be placed at start of the script (or at the end of the script)
- All subroutines have a return value (but no declaration of its type)
  - The statement
    `return value`
    can be used to terminate the execution of a subroutine and to make `value` the return value of the subroutine
  - If the execution of a subroutine terminates without encountering a `return` statement, then the value of the last evaluation of an expression in the subroutine is returned

  The return value does not have to be scalar value, but can be a list

Parameters and Arguments

Subroutines are defined as follows in Perl:

```
sub identifier {
  statements
}
```

- In Perl there is no need to declare the parameters of a subroutine (or their types)
  ⇝ there is no pre-defined fixed number of parameters
- Arguments are passed to a subroutine via a special array `@_`
- Individual arguments are accessed using `$_[0]`, `$_[1]` etc
- Is is up to the subroutine to process arguments as is appropriate
- The array `@_` is private to the subroutine
  ⇝ each nested subroutine call gets its own `@_` array

Parameters and Arguments: Examples

- The Java method

```
public static int sum2( int f, int s) {
  f = f+s;
  return f;
}
```

could be defined as follows in Perl:

```
sub sum2 {
  return $_[0] + $_[1];
}
```

- A more general solution, taking into account that a subroutine can be given arbitrarily many arguments, is the following:

```
sub sum {
  return undef if (@_ < 1);
  $sum = shift(@_);
  foreach (@_) { $sum += $_ }
  return $sum;
}
```


Private variables

```
sub sum {
  return undef if (@_ < 1);
  $sum = shift(@_);
  foreach (@_) { $sum += $_ }
  return $sum;
}
```

The variable `$sum` in the example above is global:

```
$sum = 5;
print "Value of $sum before call of sum: ",$sum,"\n";
print "Return value of sum: ",&sum(5,4,3,2,1),"\n";
print "Value of $sum after  call of sum: ",$sum,"\n";
```

produces the output

```
Value of $sum before call of sum: 5
Return value of sum: 15
Value of $sum after  call of sum: 15
```

This use of global variables in subroutines is often undesirable
⇝ we want `$sum` to be private/local to the subroutine

Private variables

- The operator `my` declares a variable or list of variables to be private:

```
my $variable;
my ($variable1,$variable2);
my @array;
```

- Such a declaration can be combined with a (list) assignment:

```
my variable = $_[0];
my (variable1,variable2) = @_;
my @array = @_;
```

- Each call of a subroutine will get its own copy of its private variables

Example:

```
sub sum {
  return undef if (@_ < 1);
  my $sum = shift(@_);
  foreach (@_) { $sum += $_ }
  return $sum;
}
```


Calling a subroutine

A subroutine is called by using the subroutine name with an ampersand `&` in front possibly followed by a list of arguments
The ampersand is optional if a list of arguments is present

```
sub identifier {
  statements
}

... &identifier  ...
... &identifier(arguments) ...
... identifier(arguments) ...
```

Examples:

```
print "sum0: ",&sum,"\n";
print "sum0: ",&sum(),"\n";
print "sum1: ",&sum(5),"\n";
print "sum2: ",&sum(5,4),"\n";
print "sum5: ",&sum(5,4,3,2,1),"\n";
$total = &sum(9,8,7,6)+&sum(5,4,3,2,1);
&sum(1,2,3,4);
```

Persistent variables

- Private variables within a subroutine are forgotten once a call of the subroutine is completed
- In Perl 5.10 and later versions, we can make a variable both private and persistent using the `state` operator
  ⇝ the value of a persistent variable will be retained between independent calls of a subroutine

Example:

```
use 5.010;

sub running_sum {
  state $sum;
  foreach (@_) { $sum += $_ }
  return $sum;
}
```


Persistent variables

Example:

```
1 use 5.010;
2
3 sub running_sum {
4   state $sum;
5   foreach (@_) { $sum += $_ }
6   return $sum;
7 }
8
9 print "running_sum():\t\t",   running_sum(),     "\n";
10 print "running_sum(5):\t",    running_sum(5),    "\n";
11 print "running_sum(5,4):\t",  running_sum(5,4),  "\n";
12 print "running_sum(3,2,1):\t",running_sum(3,2,1), "\n";
```

Output:

```
running_sum():
running_sum(5):         5
running_sum(5,4):       14
running_sum(3,2,1):     20
```


Nested subroutine definitions

- Perl allows nested subroutine definitions (unlike C or Java)

```
sub outer_sub {
   sub inner_sub { ... }
}
```

- Normally, nested subroutines are a means for information hiding
  ⇝ the inner subroutine should only be visible and executable from inside the outer subroutine
- However, Perl allows inner subroutines to be called from anywhere (within the package in which they are defined)

```
sub outer_sub {
   sub inner_sub { ... }
}
&inner_sub();
```


Nested subroutine definitions

If an inner subroutine uses a local variable of an outer subroutine, then it refers to the instance of that local variable created the first time the outer subroutine was called

Example:

```
sub outer {
  my $x = $_[0];
  sub inner { return $x }
  return inner();          # returns $_[0]?
}
print "1:␣",&outer(10),"\n";
print "2:␣",&outer(20),"\n";
```

Output:

```
1: 10
2: 10 # not 20!
```


Nested subroutine definitions: Example

```
sub sqrt2 {
  my $x = shift(@_);
  my $precision = 0.001;

  sub sqrtIter {
    my ($guess,$x) = @_;
    if (isGoodEnough($guess,$x)) {
      return int($guess/$precision+0.5)*$precision;
    } else { sqrtIter(improveGuess($guess, $x), $x) } }

  sub improveGuess {
    my ($guess,$x) = @_;
    return ($guess + $x / $guess) / 2; }

  sub isGoodEnough {
    my ($guess,$x) = @_;
    return (abs($guess * $guess - $x) < $precision); }

  return sqrtIter(1.0,$x);
}
```


Revision

Read

- Chapter 9: Processing Text with Regular Expressions
- Chapter 4: Subroutines

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.

- http://perldoc.perl.org/perlsub.html