Perl Tutorial

Diana Inkpen

School of Information Technology and Engineering

University of Ottawa

CSI 5180, Fall 2004

What is Perl

- “Practical Extraction and Report Language”.
- Created, implemented, maintained by Larry Wall.
- Interpreted language that is optimized for string manipulation, I/O, and system tasks.
- Syntax elements from the C, Bourne shell, csh, awk, sed, grep.
- (Unlike other interpreted languages, Perl compiles and optimizes the code before running.)

Some of Perl’s many strengths

- Speed of development – edit a text file and just run it. No separate compiler needed. Short pieces of code that do a lot (magic operators).
- Power – Perl’s regular expressions, easy file handling and data manipulation.
- Flexibility – freedom of expression.
- Portability – Perl compilers for all platforms.
- Extensibility – modules available on CPAN.
- Price – free. Extra: it is fun !!

The “Hello, world” program

```
#!/local/bin/perl -w
print "Hello, world!\n";
```

How to run the program written in the file *“myfile”*:

- chmod u+x myfile
- myfile

or

- perl -w myfile

Basic syntax

- Perl is free form – whitespace doesn't matter (but better use indentation for readability).
- All Perl statements end in a ; (semicolon), like C.
- Comments begin with # (pound sign) and last until the end of line (no multi-line comments).
- Variable names start with $ (or `@`, %).

More programs

-
  ```
  #!/local/bin/perl -w
  print "What is your name? ";
  $name = <>;  chomp $name;
  print "Hello, $name!\n";
  ```

-
  ```
  #!/local/bin/perl -w
  use strict;
  my $name;
  print "What is your name? ";
  $name = <>;  chomp $name;
  print "Hello, $name!\n";
  ```

Variables

- Don't need to declare variables, but better to declare them (`use strict; my $name;`).
- Cannot declare type (clear from context).
- Scalars
  - Numbers
  - Strings
- Arrays of scalars
- Hashes (associative arrays of scalars)

Strings

- Single-quoted strings.

  -
    ```
    'hello\n'               # The string 'hello\n'
    'It is 5 o \'clock!'    # ' has to be escaped
    ```
- Double-quoted strings.

  -
    ```
    "hello\n"   # The string "hello" followed by
                # new line
    ```
- Interpolation takes places only in double-quoted strings.

Scalar variables

- Hold a single scalar value (number, string, or reference).
- Begin with $, followed by letter, then possibly other letters, digits, or underscores.
-
```
use strict;
my $stuff = "Mydata"; # Assigns string to variable
$stuff = 3.5e-4;      # $stuff become 0.00035
my $things = $stuff;  # $things is now 0.00035
```

Numerical operators

-
```
use strict;
my $x = 5 * 2 + 3;         # $x is 13
my $y = 2 * $x / 4;        # $y is 6.5
my $z = (2 ** 6) ** 2;     # exponentiation
my $a = ($z - 96) * 2;     # $a is 8000
my $b = $x % 5;            # 3, 13 modulo 5
my $efg = $x-- + 5;        # $x is 12, $efg is 18
my $hij = ++$efg - --$x; # $efg is 19, $x 11, $hij 8
```

Comparison operators (1)

-
```
Operation                    Numeric      String
---------------------------------------------
less than                      <             lt
less than or equal to          <=            le
greater than                   >             gt
greater than or equal to       >=            ge
equal to                       ==            eq
not equal to                   !=            ne
compare                        <=>           cmp
```

Comparison operators (2)

-
```
use strict;
my $a = 5; my $b = 500;
$a < $b;                     # evaluates to 1
$a >= $b;                    # evaluates to ""
$a <=> $b;                   # evaluates to -1
my $c = "hello"; my $d = "there";
$d cmp $c;                   # evaluates to 1
$d ge  $c;                   # evaluates to 1
$c cmp "hello";              # evaluates to 0
```

String operators

- Concatenation (*”Hi ” . ”there”;*)

-
  ```
  use strict;
  my $greet = "Hi! ";
  my $longGreet = $greet x 2;    # value "Hi! Hi! "
  my $hi = $longGreet . "Paul."; # "Hi! Hi! Paul."
  $greet .= "All\n";             # assign and concat.
  $greet = $greet . "All\n";     # same
  ```

Variable interpolation

-
  ```
  my $s = "perl";
  my $n = 20;
  print "The string is $s and the number is $n\n";
  #prints: The string is perl and the number is 20
  print ’The string is $s and the number is $n\n’;
  #prints: The string is $s and the number is $n\n
  ```
- Interpolation only in double-quoted strings

Print: A list operator

- Accepts a list of things to print, separated by commas.
- `print "a doublequoted string ", $var, 'that was a variable called var', $num," and a newline \n";`
- `print "a doublequoted string $var that was a variable called var $num and a newline \n";`
-
```
$var="Perl"; $num=10;
print "Two \$nums are $num * 2 \n";
print "Two \$nums are ", $num * 2,"\n";
print "\$nums is ".$num."\n";
```

Arrays

- An array is an *ordered list* of scalar variables.
-
```
my @names;
@names=("Muriel","Gavin","Susanne","Sarah","Ann");
print "The elements of \@names are @names\n";
print "The first element is $names[0] \n";
print "The third element is $names[2] \n";
print 'There are ',scalar(@names)," elements\n";
print @names; print "\n";
print "@names";
print "The last element is $names[$#names]\n";
```

Functions for arrays

-
  ```
  use strict;
  my (@a, $b, $c);  @a = (1,2,3);
  push @a, 4;             # @a is (1,2,3,4)
  $b = pop @a;            # $b is 4, @a is (1,2,3)
  $b = shift @a;          # $b is 1, @a is (2,3)
  unshift @a, 5;          # @a is (5,2,3)
  $c = "one two three";
  @a = split " ",$c;      # @a is (one, two, three)
  $c = join "*", @a;
  print "$c\n";           # $c is "one*two*three"
  ```

Hashes

- A hash is an associative array.
- A hash is a list of keys that have values.

-
  ```
  my %p;
  %p =("piano" => "Susan", "basson" => "Andy");
  $p{"flute"} = "Heidi";
  $p{"oboe"} = "Jeanine";
  my (@woodwinds, @woodwindPrincipals);
  @woodwinds = keys(%p);
  @woodwindPrincipals = values(%p);
  ```

Control structures – IF (1)

-
  ```
  if (EXPRESSION) {
         STATEMENTS;
  } elsif {            # optional
         STATEMENTS;   # optional additional elsif's
  } else {             # optional
         STATEMENTS;
  }
  ```

-
  ```
  if (($a < $b) && ($c eq "hello"))
         {print "x";}
  ```

IF (2). Boolean values

-
  ```
  if ($a == $b) {print "x";}
  elsif (($a > 1) || ($b < 4)) {print "y";}
  else {print "z";}
  ```

-
  ```
  print "x" unless ($a > 4);
  unless ($a > 4) {print "x";}
  ```
- Boolean values
  - False is *0*, *””*, *”0”* , or *undef*.
  - True is anything else.

Control structures – WHILE

-
  ```
  while (EXPRESSION) {
      STATEMENTS;
  }
  ```

-
  ```
  # one way to count to 10
  $number = 0;
  while(1) {
      $number++;
      if ($number >= 10 )  { last; }
  }
  ```
- *last;* and *next;*

Control structures – FOR

-
  ```
  for ( INITIAL_EXPR ; COND_EXPR ; LOOP_EXPR ) {
      STATEMENTS;
  }
  ```

-
  ```
  # an example countdown
  for ( $i=10 ; $i>=0 ; $i-- ) {
           print("$i\n");
  }
  ```

-
  ```
  # printing the array @a
  for ( $i=0 ; $i <= $#a ; $i++ )
     { print("$a[$i]\n"); }
  ```

Control structures – FOREACH

-
  ```
  foreach SCALAR ( LIST ) {
      STATEMENTS;
  }
  ```

-
  ```
  my $k;
  foreach $k (@a)
     { print "$k\n"; }
  ```

-
  ```
  # prints a hash
  foreach $k (keys % p)
     { print "$k  $p{$k}\n"; }
  ```

Basic I/O

-
  ```
  my $line;
  # this program reads from SDTIN (until ^D)
  # or from the file specified in the command line
  while($line = <>)  { print $line; }
  my $x = $ARGV[0];
  if ($x ne "") {print "The filename was $x\n";}
  ```

-
  ```
  $a = <>; # reads a line
  @b = <>; # reads all lines of the file
  ```

-
  ```
  # formatted output with printf
  printf "%15s %5d %10.2f\n", $s, $n, $r;
  ```

Defaults

- $_ is the name of the default scalar variable
-
```
foreach (keys % p)
   { print "$_  $p{$_}\n"; }
```
-
```
while(<>)
{ print $_; }
```
-
```
while(<>)
{ print; }
```

Regular expressions (R.E.) (1)

- A regular expression describes a set of possible strings.
-
```
# prints only lines containing "abc"
my $line;
while ($line = <>){
 if ($line =~ /abc/) { print $line; }
}
```
-
```
my $line;
while (<>){
 if (/abc/) { print $_; }
}
```

R.E. (2) – Single-character matching

-
  ```
  $line =~ /[abc]/;  # a, b, or c
  $line =~ /[0-9]/;  # one digit
  $line =~ /\d/;     # one digit
  $line =~ /[^A-Z]/; # one character not in A-Z
  $line =~ /a.c/;    # a, any character except \n, c
  ```

-
  ```
  \d   (a digit, equivalent to [0-9])
  \w   (word char, equivalent to [a-zA-Z0-9_])
  \s   (space char, equivalent to [ \r\t\n\f])
  \D  \W  \S   negations of the previous
  ```

R.E. (3) – Grouping patterns

- Sequence (/abc/)
- Alternation (/a|b|c/, /(aa)|(bb)/)
- Multiplication

  -
    ```
    if ($l=~/ab*c/) {print $l;}  # a, 0 or more b, c
    if ($l=~/ab+c/) {print $l;}  # a, 1 or more b, c
    ```

  -
    ```
    *      0 or more
    +      1 or more
    ?      0 or 1     ( /a?bc/ matches bc and abc)
    *?     0 or more - the shortest match
    {2,5}  between 2 and 5 occurences
    ```

Pattern-matching operators (4)

- not match operator `$line !~ /ab*c/i;`

-
  ```
  i  case-insensitive pattern matching
  m  treat string as multiple lines (^ and $ match \n)
  s  treat string as single lines (^ and $
     ignore \n, but . matches \n)
  x  extend legibility with spaces and comments
  o  only compile pattern once
  ```
- interpolation `$line =~ /$arg/o;`

R. E. (5) – Anchoring patterns

-
  ```
  /fred\b/ matches fred but not frederick
  /a(?=xx)/ matches a if followed by xx
  ```

-
  ```
  ^     matches at beginning of string (line if /m)
  $     matches at end of string (line if /m used)
  \b    matches at word boundary
  \B    matches except word boundary
  \A    matches at beginning of string
  \Z    matches at end of string
  (?=...) matches if ... would match next
  (?!...) matches if ... wouldn't match next
  ```

R. E. (6) – Returning matches

-
  ```
  $line = "aBCdefg";
  ($a,$b)=($line=~/(abc)de(fg)/io);
  ```

-
  ```
  $line =~ /(abc)de(fg)/io;
  $a = $1; $b = $2;     # $a is aBC, $b is fg
  ```

-
  ```
  $line =~ /.e/io;
  $pre = $‘;        # $pre is aBC
  $match = $&;      # $match is de
  $post = $’;       # $post is fg
  ```

Substitutions

- `/pattern/   m/pattern/   ?pattern?`
- `s/pattern/replacement/modifiers`

-
  ```
  $line = "aabaaab";
  $line =~ s/a+/x/go;   # $line is "xbxb"
  $_ = "foot fool buffoon";
  s/foo/bar/g;          # $_ is "bart barl bufbarn";
  s/(\w+)/<$1>/g; # $_ is "<bart> <barl> <bufbarn>";
  ```

Translations

- `tr/searchlist/replacementlist/modifiers`

-
  ```
  $line = "xyz";
  $line =~ tr/a-z/A-Z/; # $line is XYZ
  $_ = "abAB12";
  $_ =~ tr/A-Z/a-z/;    # $_ is abab12
  tr/0-9/x/;            # $_ is ababxx
  $cnt = tr/a/a/;       # $cnt is 2, $_ is ababxx
  $cnt = s/a/a/g;       # $cnt is 2
  ```

Subroutines (1)

- `sub greet { print "Hello\n"; }`

-
  ```
  sub add_two
  {  my $a = shift;  my $b = shift;
     return $a + $b;
  }
  ```

-
  ```
  sub add
  { my @a = @_; my ($i, $sum); $sum = 0;
    foreach $i (@a) { $sum += $i}
    return $sum;
  }
  ```

Subroutines (2)

-
  ```
  &greet;                # "Hello"
  greet;                 # "Hello"
  print add_two(2,3);    # 5
  print add(1,2,3,4);    # 10
  ```

-
  ```
  sub add
  { my $sum = 0;
    foreach  (@_) { $sum += $_}
    return $sum;
  }
  ```

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

Filehandles

- *STDIN, STDOUT, STDERR*

-
  ```
  my $line;
  open(A, "<addr") || die "Cannot open addr $!";
  open(F, ">ofile") || die "Cannot open ofile $!";
  while ($line = <A>)
   { $line =~ tr/[A-Z]/[a-z]/;
     print F $line;
   }
  close(A); close(F);
  ```

Formats (1)

-
  ```
  # define the format
  format ADRLABEL =
  =============================
  | @<<<<<<<<<<<<<<<<<<<<<<<< |
  $name
  | @<<<<<<<<<<<<<<<<<<<<<<<< |
  $address
  | @<<<<<<<<<<<<<<< @< @<<<< |
  $city, $state, $zip
  =============================
  .
  ```

-
  ```
  Stonehenge:4470 SW Hall suite 107:Beaverton:OR:97005
  Fred Flinstone:3737 Hard Rock Lane:Bedrock:OZ:999bc
  ```

Formats (2)

-
  ```
  open(ADRLABEL,">labels") || die "Cannot create labels $!";
  open(A, "addr") || die "Cannot open addr $!";
  while (<A>)
   { chomp;                           # load data
    ($name, $address, $city, $state, $zip) = split /:/;
     write (ADRLABEL);  # invoke the format
   }
  close(ADRLABEL); close(A);
  ```

-
  ```
  =============================
  | Stonehenge                |
  | 4470 SW Hall suite 107    |
  | Beaverton        OR 97005 |
  =============================
  ```

Context considerations

```
Expression Context     Evaluates to
----------------------------------------------------------------------
$scalar    scalar      the value held in $scalar
@array     list        the list of values (in order) held in @array
@array     scalar      the total number of elements in @array
                       (same as $#array + 1)
$array[$x] scalar      the $xth element of @array
$#array    scalar      the subscript of the last element in @array
"$scalar"  scalar(interp.)  a string containing the content of $scalar
"@array"   scalar(interp.)  a string containing the elements of @array,
                            separated by spaces
%hash      list        a list of alternating keys and values from %hash
$hash{$x}  scalar      the element from %hash with the key of $x
```

Command line arguments

-
  ```
  use strict;
  use Getopt::Std;
  use  vars qw($opt_s $opt_f $opt_o);
  $opt_s = 0; $opt_f = ""; $opt_o="";
  if ( ! getopts('f:so:')) {
  print STDERR "Usage: $0 [-f inputfile]",
   "[-s] [-o outputfile]\n";  exit;
  }
  ```

-
  ```
  my_program -s -f addr -o labels
  # $opt_f is "addr", $opt_o is "labels", $opt_s is 1
  ```

Optimization for speed

- Perl procedure calls are expensive, especially if you're passing in arrays or hashes. So feel free to inline subroutines called within a tight loop. It's not as nice a programming style, but the performance gain is worth it. You should put comments in the code, however, indicating where you're doing this and that speed gain is the reason for it.
- Use hashes instead of arrays when possible.

Perl hints – to be used in A1 to A3

- Write readable code! Perl will let you write cryptic code, but Perl code does not have to be cryptic.
- Use the -*w* flag. It will give you many warnings that almost always point out errors in your code. (but slower)
- Use *"use strict;"* at the beginning of your code. It forces you to declare your variables and gives you error messages which, once fixed, make your code easier to read.
- Read Ch.8 of Programming Perl, Common Goofs for Novices, Efficiency and Programming with Style.

References

- Schwartz, Randal L. and Christiansen, Tom, Learning Perl, second edition, O’Reilly, 1997.
- Wall, Larry; Christiansen, Tom; and Schwartz, Randal L., Programming Perl, third edition, O’Reilly, 2000.
- Perl Tutorial http://www.ncsa.uiuc.edu/General/Training/PerlIntro/
- Introduction to Perl http://www.cclabs.missouri.edu/things/instruction/perl/perlcourse.html
- Picking Up Perl, a Freely Redistributable Perl Tutorial Book http://www.ebb.org/PickingUpPerl/