Perl

What is Perl?

- Practical Extraction and Report Language
- Scripting language created by Larry Wall in the mid-80s
- Functionality and speed somewhere between low-level languages (like C) and high-level ones (like “shell”)
- Influence from awk, sed, and C Shell
- Easy to write (after you learn it), but sometimes hard to read
- Widely used in CGI scripting

A Simple Perl Script

hello:

turns on warnings

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

```
$ chmod a+x hello
$ ./hello
Hello, world!
$ perl -e 'print "Hello, world!\n"'
Hello, world!
```

Data Types

- Type of variable determined by special leading character

`$foo`	scalar
`@foo`	list
`%foo`	hash
`&foo`	function

- Data types have separate name spaces

Scalars

- Can be numbers

```
$num = 100;                    # integer
$num = 223.45;             # floating-point
$num = -1.3e38;
```

- Can be strings

```
$str = ‘good morning’;
$str = “good evening\n”;
$str = “one\ttwo”;
```

- *Backslash escapes* and variable names are interpreted inside double quotes
- No boolean data type: 0 or ‘’ means false
  - ! negates boolean value

Special Scalar Variables

$0	Name of script
$_	Default variable
$$	Current PID
$?	Status of last pipe or system call
$!	System error message
$/	Input record separator
$.	Input record number
undef	Acts like 0 or empty string

Operators

- Numeric: **+ - * / % ****
- String concatenation: **.**

```
$state = “New” . “York”;    # “NewYork”
```

- String repetition: **x**

```
print “bla” x 3;        # blablabla
```

- Binary assignments:

```
$val = 2; $val *= 3;    # $val is 6
$state .= “City”;       # “NewYorkCity”
```

Comparison Operators

Comparison	Numeric	String
Equal	==	eq
Not Equal	!=	ne
Greater than	<	lt
Less than or equal to	>	le
Greater than or equal to	>=	ge

undef and defined

```
$f = 1;
while ($n < 10) {
   # $n is undef at 1st iteration
   $f *= ++$n;
}
```

- Use defined to check if a value is **undef**

```
if (defined($val)) { … }
```

Lists and Arrays

- List: ordered collection of scalars
- Array: Variable containing a list
- Each element is a scalar variable
- Indices are integers starting at 0

Array/List Assignment

```
@teams=(“Knicks”,“Nets”,“Lakers”);
print $teams[0];      # print Knicks
$teams[3]=“Celtics”; # add new elt
@foo = ();            # empty list
@nums = (1..100);    # list of 1-100
@arr = ($x, $y*6);
($a, $b) = (“apple”, “orange”);
($a, $b) = ($b, $a); # swap $a $b
@arr1 = @arr2;
```

More About Arrays and Lists

- Quoted words - **qw**

```
@planets = qw/ earth mars jupiter /;
@planets = qw{ earth mars jupiter };
```

- Last element's index: $#planets
  - Not the same as number of elements in array!
- Last element: $planets[-1]

Scalar and List Context

```
@colors = qw< red green blue >;
```

- Array as string:

```
print “My favorite colors are @colors\n”;
```

  - Prints My favorite colors are red green blue

- Array in scalar context returns the number of elements in the list

```
$num = @colors + 5;        # $num gets 8
```

- Scalar expression in list context

```
@num = 88;          # one element list (88)
```

pop and push

- **push** and **pop**: arrays used as stacks
- **push** adds element to end of array

```
@colors = qw# red green blue #;
push(@colors, “yellow”);      # same as
@colors = (@colors, “yellow”);
push @colors, @more_colors;
```

- **pop** removes last element of array and returns it

```
$lastcolor = pop(@colors);
```

shift and unshift

- **shift** and **unshift**: similar to push and pop on the "left" side of an array
- **unshift** adds elements to the beginning

```
@colors = qw# red green blue";
unshift @colors, "orange";
```

  - First element is now "orange"
- **shift** removes element from beginning

```
$c = shift(@colors); # $c gets "orange"
```

sort and reverse

- **reverse** returns list with elements in reverse order

```
@list1 = qw# NY NJ CT #;
@list2 = reverse(@list1); # (CT,NJ,NY)
```

- **sort** returns list with elements in ASCII- sorted order

```
@day = qw/ tues wed thurs /;
@sorted = sort(@day); #(thurs,tues,wed)
@nums = sort 1..10; # 1 10 2 3 … 8 9
```

- **reverse** and **sort** do not modify their arguments
- **reverse** in scalar context flip characters in string

```
$flipped = reverse(“abc”); # gets “cba”
```

Iterate over a List

- **foreach** loops throught a list of values

```
@teams = qw# Knicks Nets Lakers #;
foreach $team (@teams) {
  print “$team win\n”;
}
```

- Value of *control variable* is restored at the end of the loop
- **$_** is the default

```
foreach (@teams) {
  $_ .= “ win\n”;
  print;                          # print $_
}
```

Hashes

- Associative arrays - indexed by strings (keys)

```
$cap{“Hawaii”} = “Honolulu”;
%cap = ( “New York”, “Albany”, “New Jersey”,
  “Trenton”, “Delaware”, “Dover” );
```

- Can use **=>** (the *big arrow* or *comma arrow*) in place of **,**

```
%cap = ( “New York”   => “Albany”,
         “New Jersey” => “Trenton”,
         Delaware     => “Dover” );
```

Hash Element Access

- ***$hash{$key}***

```
print $cap{“New York”};
print $cap{“New “ . “York”};
```

- Unwinding the hash

```
@cap_arr = %cap;
```

  – Gets unordered list of key-value pairs

- Assigning one hash to another

```
%cap2 = %cap;
%rev_cap = reverse %cap;
print $rev_cap{“Trenton”}; # New Jersey
```

Hash Functions

- **keys** returns a list of keys

```
@state = keys %cap;
```

- **values** returns a list of values

```
@city = values %cap;
```

- Use **each** to iterate over all (key, value) pairs

```
while ( ($state, $city) = each %cap )
{
  print “Capital of $state is $city\n”;
}
```

Subroutines

- **sub *myfunc* { … }**

```
$name="Jane";
…
sub print_hello {
  print "Hello $name\n"; # global $name
}
&print_hello;  # print "Hello Jane"
print_hello;         # print "Hello Jane"
hello();       # print "Hello Jane"
```

Arguments

- Parameters are assigned to the special array @_
- Individual parameter can be accessed as $_[0], $_[1], …

```
sub sum {
  my $x;          # private variable $x
  foreach (@_) {      # iterate over params
   $x += $_;
  }
  return $x;
}
$n = &sum(3, 10, 22);        # n gets 35
```

More on Parameter Passing

- Any number of scalars, lists, and hashes can be passed to a subroutine
- Lists and hashes are "flattened"

```
func($x, @y, %z);
```

  - Inside `func`:
    - `$_[0]` is `$x`
    - `$_[1]` is `$y[0]`
    - `$_[2]` is `$y[1]`, etc.
- The scalars in `@_` are implicit aliases (not copies) of the ones passed, i.e. changing the values of `$_[0]`, etc. changes the original variables

Return Values

- The return value of a subroutine is the last expression evaluated, or the value returned by the **return** operator

```
sub myfunc {
  my $x = 1;
  $x + 2; #returns 3
}
```

```
sub myfunc {
   my $x = 1;
   return $x + 2;
}
```

- Can also return a list: `return @somelist;`
- If **return** is used without an expression (failure), **undef** or **()** is returned depending on context

Lexical Variables

- Variables can be scoped to the enclosing block with the **my** operator

```
sub myfunc {
  my $x;
  my($a, $b) = @_;    # copy params
  …
}
```

- Can be used in any block, such as an if block or while block
  - Without enclosing block, the scope is the source file

Another Subroutine Example

```
@nums = (1, 2, 3);
$num = 4;
@res = dec_by_one(@nums, $num); # @res=(0, 1, 2, 3)
                                # (@nums,$num)=(1, 2, 3, 4)
dec_by_1(@nums, $num);          # (@nums,$num)=(0, 1, 2, 3)

sub dec_by_one {
    my @ret = @_;               # make a copy
    for my $n (@ret) { $n-- }
    return @ret;
}
sub dec_by_1 {
    for (@_) { $_-- }
}
```

Reading from STDIN

- STDIN is the builtin filehandle to the standard input
- Use the line input operator around a file handle to read from it

```
$line = <STDIN>;     # read next line
chomp($line);
```

- **chomp** removes trailing string that corresponds to the value of $/ - usually the newline character

Reading from STDIN example

```
while (<STDIN>) {
  chomp;
  print “Line $. ==> $_\n”;
}
```

```
Line 1 ==> [Contents of line 1]
Line 2 ==> [Contents of line 2]
…
```

<>

- The *diamond operator* <> makes Perl programs work like standard Unix utilities
- Lines are read from list of files given as command line arguments (@ARGV)

```
while (<>) {
  chomp;
  print "Line $. from $ARGV is $_\n";
}
```

- `./myprog file1 file2 -`
  - Read from file1, then file2, then standard input
- $ARGV is the current filename

Filehandles

- Use open to open a file for reading/writing

```
open LOG, “syslog”;      # read
open LOG, “<syslog”;     # read
open LOG, “>syslog”;     # write
open LOG, “>>syslog”;    # append
```

- Close a filehandle after using the file

```
close LOG;
```

Errors

- When a fatal error is encountered, use **die** to print out error message and exit program

```
die “Something bad happened\n” if ….;
```

- Always check return value of **open**

```
open LOG, “>>syslog”
  or die “Cannot open log: $!”;
```

- For non-fatal errors, use **warn** instead

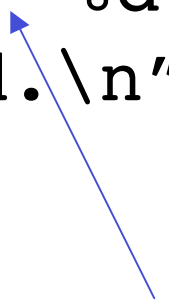
```
warn “Temperature is below 0!”
  if $temp < 0;
```

Reading from a File

```
open MSG, “/var/log/messages”
   or die “Cannot open messages: $!\n”;
while (<MSG>) {
   chomp;
   # do something with $_
}
close MSG;
```

Writing to a File

```
open LOG, ">/tmp/log"
  or die "Cannot create log: $!";
print LOG "Some log messages…\n"
printf LOG "%d entries
  processed.\n", $num;
close LOG;
```



no comma after filehandle

Manipulating Files and Dirs

- **unlink** removes files

```
unlink “file1”, “file2”
   or warn “failed to remove file: $!”;
```

- **rename** renames a file

```
rename “file1”, “file2”;
```

- **link** creates a new (hard) link

```
link “file1”,  “file2”
   or warn “can’t create link: $!”;
```

- **symlink** creates a soft link

```
link “file1”, “file2” or warn “ … “;
```

Manipulating Files and Dirs cont.

- **mkdir** create directory

```
mkdir “mydir”, 0755
  or warn “Cannot create mydir: $!”;
```

- **rmdir** remove empty directories

```
rmdir “dir1”, “dir2”, “dir3”;
```

- **chmod** modifies permissions on a file or directory

```
chmod 0600, “file1”, “file2”;
```

if - elsif - else

- **if** … **elsif** … **else** …

```
if ( $x > 0 ) {
  print “x is positive\n”;
}
elsif ( $x < 0 ) {
  print “x is negative\n”;
}
else {
  print “x is zero\n”;
}
```

while and until

```
while ($x < 100) {
  $y += $x++;
}
```

- until is like the opposite of while

```
until ($x >= 100) {
  $y += $x++;
}
```

for

- for (*init*; *test*; *incr*) { … }

```
# sum of squares of 1 to 5
for ($i = 1; $i <= 5; $i++) {
  $sum += $i*$i;
}
```

next

- **next** skips the remaining of the current iteration (like continue in C)

```
# only print non-blank lines
while (<>) {
  if ( $_ eq “\n”) { next; }
  else { print; }
}
```

last

- **last** exist the loop immediately (like break in C)

```
# print up to first blank line
while (<>) {
  if ( $_ eq “\n”) { last; }
  else { print; }
}
```

Logical AND/OR

- Logical AND : **&&**

```
if (($x > 0) && ($x < 10)) { … }
```

- Logical OR : ||

```
if  ($x < 0) || ($x > 0)) { … }
```

- Both are short-circuit operators - the second expression is only evaluated if necessary

Regular Expressions

- Use EREs (egrep style)
- Plus the following character classes
  - \w "word" character: `[A-Za-z0-9_]`
  - \d digits: `[0-9]`
  - \s whitespace: `[\f\t\n\r ]`
  - \b word boundary
  - \W, \D, \S, \B are complements of the corresponding classes above
- Can use \t to denote a tab

Backreferences

- Support backreferences
- Subexpressions are referred to using \1, \2, etc. in the RE and $1, $2, etc. outside the RE

```
if (/^this (red|blue|green) (bat|ball) is \1/)
{
  ($color, $object) = ($1, $2);
}
```

Matching

- Pattern match operator: **/RE/** is a shortcut of **m/RE/**
  - Returns true if there is a match
  - Match against **$_** be default
  - Can also use **m(RE)**, **m<RE>**, **m!RE!**, etc.

```
if (/^\/usr\/local\//) { … }
if (m%/usr/local/%) { … }
```

- Case-insensitive match

```
if (/new york/i) { … };
```

Matching cont.

- To match an RE against something other than $_, use the *binding operator* =~

```
if ($s =~ /\bblah/i) {
  print “Find blah!”
}
```

- !~ negates the match

```
while (<STDIN> !~ /^#/) { … }
```

- Variables are interpolated inside REs

```
if (/^$word/) { … }
```

Match Variables

- Special match variables
  - `$&` : the section matched
  - `$`` : the part before the matched section
  - `$'` : the part after the matched section

```
$string = "What the heck!";
$string =~ /\bt.*e/;
print "($`) ($&) ($')\n";
(What ) (the he) (ck!)
```

Substitutions

- Sed-like search and replace with **s///**

```
s/red/blue/;
$x =~ s/\w+$/$`/;
```

  – Unlike **m///**, **s///** modifies the variable

- Global replacement with **/g**

```
s/(.)\1/$1/g;
```

- Transliteration operator: **tr///** or **y///**

```
tr/A-Z/a-z/;
```

RE Functions

- **split** string using RE (whitespace by default)

```
@fields = split /:/, "::ab:cde:f";
# gets ("","","ab","cde","f")
```

- **join** strings into one

```
$str = join "-", @fields;  # gets "--ab-cde-f"
```

- **grep** something from a list
  - Similar to UNIX grep, but not limited to using regular expressions

```
@selected = grep(!/^#/, @code);
```

  - Modifying elements in returned list actually modifies the elements in the original list

Running Another program

- Use the system function to run an external program
- With one argument, the shell is used to run the command
  - Convenient when redirection is needed

```
$status = system("cmd1 args > file");
```

- To avoid the shell, pass system a list

```
$status = system($prog, @args);
die "$prog exited abnormally: $?" unless
  $status == 0;
```

Capturing Output

- If output from another program needs to be collected, use the backticks

```
my $files = `ls  *.c`;
```

  - Collect all output lines into a single string

```
my @files = `ls *.c`;
```

  - Each element is an output line
- The shell is invoked to run the command

Environment Variables

- Environment variables are stored in the special hash %ENV

```
$ENV{‘PATH’} =
  “/usr/local/bin:$ENV{‘PATH’}”;
```

Example: Union and Intersection I

```
@a = (1, 3, 5, 6, 7);
@b = (2, 4, 5, 9);
@union = @isect = ();
%union = %isect = ();

foreach $e (@a) { $union{$e} = 1}
foreach $e (@b) {
  if ($union($e) ) { $isect{$e} = 1 }
  $union{$e} = 1;
}
@union = keys %union;
@isect = keys %isect;
```

Example: Union and Intersection II

```
@a = (1, 3, 5, 6, 7);
@b = (2, 4, 5, 9);
@union = @isect = ();
%union = %isect = ();

foreach $e (@a, @b) {
  $union{$e}++ && $isect{$e}++;
}
@union = keys %union;
@isect = keys %isect;
```

Example: Word Frequency

```
#!/usr/bin/perl -w
# Read a list of words (one per line) and
# print the frequency of each word
use strict;
my(@words, %count, $word);
chomp(@words = <STDIN>);  # read and chomp all lines
foreach $word (@words) {
  $count{$word} += 1;
}
foreach $word (keys %count) {
  print “$word was seen $count{$word} times.\n”;
}
```

Good Ways to Learn Perl

- **a2p**
  - Translates an **awk** program to Perl
- **s2p**
  - Translates a **sed** script to Perl
- **perldoc**
  - Online perl documentation

```
$ perldoc perldoc <-- perldoc man page
$ perldoc -f sort <-- Perl sort function man page
$ perldoc CGI     <-- CGI module man page
```

Modules

- Perl modules are libraries of reusable code with specific functionalities
- Standard modules are distributed with Perl, others can be obtained from 

- Include modules in your program with use, e.g. `use CGI` incorporates the CGI module
- Each module has its own namespace

Perl CGI Module

- Interface for parsing and interpreting query strings passed to CGI scripts
- Methods for creating generating HTML
- Methods to handle errors in CGI scripts
- Two interfaces: procedural and object-oriented
  - Need to ask for the procedural interface

```
use CGI qw(:standard);
```

A (rather ugly) CGI Script

```
#!/usr/bin/perl

$size_of_form_info = $ENV{'CONTENT_LENGTH'};
read ($STDIN, $form_info, $size_of_form_info);

# Split up each pair of key/value pairs
foreach $pair (split (/&/, $form_info)) {
   # For each pair, split into $key and $value variables
   ($key, $value) = split (/=/, $pair);
   # Get rid of the pesky %xx encodings
   $key =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
   $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
   # Use $key as index for $parameters hash, $value as value
   $parameters{$key} = $value;
}

# Print out the obligatory content type line
print "Content-type: text/plain\n\n";

# Tell the user what they said
print "Your birthday is on " . $parameters{birthday} . ".\n";
```

A Perl CGI Script

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

use strict;
use CGI qw(:standard);

my $bday = param("birthday");

# Print headers (text/html is the default)
print header(-type => 'text/html');
# Print <html>, <head>, <title>, <body> tags etc.
print start_html("Birthday");
# Your HTML body
print p("Your birthday is $bday.");
# Print </body></html>
print end_html();
```

- Read the CGI Perl documentation (`perldoc CGI`)

Further Reading