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

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

```bash
$ 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$foo</td>
<td>scalar</td>
</tr>
<tr>
<td>@foo</td>
<td>list</td>
</tr>
<tr>
<td>%foo</td>
<td>hash</td>
</tr>
<tr>
<td>&amp;foo</td>
<td>function</td>
</tr>
</tbody>
</table>

• 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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Name of script</td>
</tr>
<tr>
<td>$._</td>
<td>Default variable</td>
</tr>
<tr>
<td>$$_</td>
<td>Current PID</td>
</tr>
<tr>
<td>$?</td>
<td>Status of last pipe or system call</td>
</tr>
<tr>
<td>$!</td>
<td>System error message</td>
</tr>
<tr>
<td>$/</td>
<td>Input record separator</td>
</tr>
<tr>
<td>$.</td>
<td>Input record number</td>
</tr>
<tr>
<td><code>undef</code></td>
<td>Acts like 0 or empty string</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Numeric</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td><code>==</code></td>
<td><code>eq</code></td>
</tr>
<tr>
<td>Not Equal</td>
<td><code>!=</code></td>
<td><code>ne</code></td>
</tr>
<tr>
<td>Greater than</td>
<td><code>&lt;</code></td>
<td><code>lt</code></td>
</tr>
<tr>
<td>Less than or equal to</td>
<td><code>&gt;</code></td>
<td><code>le</code></td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td><code>&gt;=</code></td>
<td><code>ge</code></td>
</tr>
</tbody>
</table>
$f = 1;
while ($n < 10) {
   # $n is undef at 1st iteration
   $f *= ++$n;
}

• Use defined to check if a value is undefined
  if (defined($val)) { ... }

undef and defined
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`
  
  ```perl
  @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
  ```perl
  @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
  ```perl
  $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
  
  ```perl
  @colors = qw# red green blue#
  unshift @colors, "orange";
  ```

  - First element is now “orange”

- **shift** removes element from beginning
  
  ```perl
  $c = shift(@colors); # $c gets “orange”
  ```
sort and reverse

- **reverse** returns list with elements in reverse order
  ```perl
  @list1 = qw# NY NJ CT #;
  @list2 = reverse(@list1); # (CT,NJ,NY)
  ```

- **sort** returns list with elements in ASCII- sorted order
  ```perl
  @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
  ```perl
  $flipped = reverse("abc"); # gets "cba"
  ```
Iterate over a List

- **foreach** loops throught a list of values
  ```perl
  @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
  ```perl
  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

• \texttt{$hash\{key\}$}
  \begin{verbatim}
  print $cap{“New York”};
  print $cap{“New “ . “York”};
  \end{verbatim}

• Unwinding the hash
  \begin{verbatim}
  @cap_arr = %cap;
  \end{verbatim}
  – Gets unordered list of key-value pairs

• Assigning one hash to another
  \begin{verbatim}
  %cap2 = %cap;
  %rev_cap = reverse %cap;
  print $rev_cap{“Trenton”};  # New Jersey\end{verbatim}
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 { ... }`
  ```perl
  $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], ...

```perl
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”
  ```perl
  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
  }

• 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
  ```perl
  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 = \texttt{dec\_by\_one}(@nums, $num);  \quad \# \res=(0, 1, 2, 3)
\quad \# (@nums,\$num)=(1, 2, 3, 4)
dec\_by\_1(@nums, $num); \quad \# (@nums,\$num)=(0, 1, 2, 3)

\begin{verbatim}
sub \texttt{dec\_by\_one} { 
  my @ret = @_; \quad \# make a copy
  for my $n (@ret) { $n-- }
  return @ret;
}
sub dec\_by\_1 { 
  for (@_) { $_[-- }
}
\end{verbatim}
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
  ```perl
  $line = <STDIN>;    # read next line
  chomp($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)

  ```perl
  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: $!

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...
printf LOG "%d entries processed.\n", $num;
close LOG;

no comma after filehandle
Manipulating Files and Dirs

- **unlink** removes files
  
  ```bash
  unlink "file1", "file2"
  
  or warn "failed to remove file: $!";
  ```

- **rename** renames a file
  
  ```bash
  rename "file1", "file2"
  ```

- **link** creates a new (hard) link
  
  ```bash
  link "file1", "file2"
  
  or warn "can't create link: $!";
  ```

- **symlink** creates a soft link
  
  ```bash
  link "file1", "file2" or warn " ... ";
  ```
Manipulating Files andDirs 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

```php
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)

```perl
# 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

```perl
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 \textit{cont.}

- To match an RE against something other than $\_\_\_$, use the \textit{binding operator} $=~$
  
  \begin{verbatim}
  if ($s =~ /\bblah/i) {
    print "Find blah!"
  }
  \end{verbatim}

- $!~$ negates the match
  
  \begin{verbatim}
  while (<STDIN> !~ /^#/) { ... }
  \end{verbatim}

- Variables are interpolated inside REs
  
  \begin{verbatim}
  if (/^$word/) { ... }
  \end{verbatim}
Match Variables

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

$string = "What the heck!";
$string =~ /\bt.*e/;
print "($`) ($&) ($')
(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)
  ```perl
  @fields = split /:/, "::ab:cde:f";
  # gets ("", ",", "ab", "cde", "f")
  ```
- **join** strings into one
  ```perl
  $str = join "-", @fields;  # gets "--ab-cde-f"
  ```
- **grep** something from a list
  ```perl
  @selected = grep (!/^#/), @code);
  ```
  - Similar to UNIX grep, but not limited to using regular expressions
  - 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) {  # read and chomp all lines
    $count{$word} += 1;
}
foreach $word (keys %count) {  # read and chomp all lines
    print "$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 CPAN
• 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

```perl
#!/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