Perl

- is an imperative language
- supports many programming styles (including object-oriented)
- is portable across platforms

"A language for getting your job done!"
—Larry Wall
Designer and primary implementor of Perl

Design philosophy

- No a priori design, no committees!
- A mix-and-match accumulation of useful features desired by real programmers over many years
- Originally for text processing, generating reports
- Has features from C, Java, Unix shells, awk, and sed

What is it used for?

- Text processing, generating reports
- GUI front-ends to command-line commands
- Systems integration programming
- Web CGI scripting
- ...and lots lots more!

Perl is incredibly useful

Of about 480 of my general-purpose scripts
- ~ 220 are sh shell scripts
  (many of these use Perl inside!)
- ~ 130 are zsh shell scripts
- ~ 110 are Perl scripts

Perl language features

- Dynamically typed
- Lexical and dynamic scoping
- First-class functions
- Built-in arrays, lists, hash-tables, "regular expressions"
- Module system
- Automatic memory reclamation (via reference counting)
Sample task

Print a report of the users of a given computer system using /etc/passwd
• Input: /etc/passwd file
• Output: Human readable report

Think about how you would do this in C++ or Java...

Running Perl code

• Can use shebang (sharp-bang) lines when running under Unix varieties:
  #!/usr/bin/perl
  means to use the binary "/usr/bin/perl" to interpret the remaining lines of the file
• Can also run Perl directly:
  perl passwd-report

Ultra-fast byte-compilation

• Perl seems to be interpreted—very fast turnaround time
• Actually, it byte-compiled the source code very quickly, saves the byte-codes in memory, and then has a virtual machine that runs those byte codes
• Fast compilation + surprisingly fast execution = easy and quick development

Perl philosophies

• There's more than one way to do it (TMTOWTDI)
• The long term lazy way
  Do it right, since you'll end up using it over and over again
• 3 great virtues of a programmer: Laziness, impatience, and hubris

Conditionals in Perl

if ($nLines < 0) {
  $nLines = 0;
}  
  
Focus on the conditional?  
or  
  
Focus on the assignment?

or

$nLines = 0 if $nLines < 0;

Larry Wall is a linguist

If you make a cup of tea,
  I'll drink it.
or
  
I'll drink a cup of tea if you make it.
Variables

- $scalar: number, string, reference
- @array: heterogeneous
- %hash: maps keys to values
- &subroutine: usually omit the &

Comparisons

- < == > for comparing numbers
- lt eq gt for comparing strings

"a" < "b" ⇒ undef
"a" lt "b" ⇒ 1
"11" < "2" ⇒ undef
"11" lt "2" ⇒ 1  Interpreted as TRUE

Strings and numbers are one and the same

"11" < 2 ⇒ undef
"11" lt 2 ⇒ 1

- Instead of giving a type error, Perl is defined to give a reasonable
  meaning to virtually any expression!
- Downside: sometimes the meaning may
  surprise or confuse you!

Variable interpolation and string literals

```
my $a = 2;
my $b = "World";
print STDOUT "Hello $b
1+1=$a
";
```

Output:
Hello World
1+1=2

```
my $a = 2;
my $b = "World";
print STDOUT "Hello $b\n1+1=$a\n";
```

Arrays and lists

```
my @names = split(/,/,"jill,bob,sam");
my @colors = ("red","green","blue");
$colors[0] ⇒ "red"
$colors[1] ⇒ "NewColor"
join(",",@colors) ⇒ "red, GREEN, blue"
```

Hash tables

```
%longday = (
  "Sun"  => "Sunday",
  "Mon"  => "Monday",
  "Tue"  => "Tuesday",
  ...
  "Sat"  => "Saturday",
);
$longday{"Mon"} ⇒ "Monday"
```

Not a typo—trailing comma is ignored and makes editing easier!
Iteration

for my $d (values % longday) {
    print $d, "\n";
}

But could just write:
print join("\n", (values % longday)), "\n";

Iteration and lists

@longday_vals = values % longday;
foreach my $d (@longday_vals) {
    print $d, "\n";
}
while (my $arg = shift @ARGV) {
    print "$arg\n";
}

Reading from files

# Print all lines from standard input
# that contain the substring "greg"
while (<>) {
    print if / greg/;
}

Lots of magic here—reads from standard input, and assigns to $_
More magic—as if we wrote:
print $_ if ($_ =~ m/greg/)

Regular expressions

• Very powerful “wildcard-like” tool
• Simple cases, just matching substrings
  "Hi Greg, how are you" =~ m/ greg/ ⇒ undef
  "Hi Greg, how are you" =~ m/Greg/ ⇒ 1
  "Hi Greg, how are you" =~ m/greg/i ⇒ 1

Regular-expression control flag:
  Ignore case!

Regular expression
meta-characters

• Matches any character (except newline)
"Hi Greg, how are you" =~ m/G./ ⇒ 1
"Hi Greg, how are you" =~ m/G.e/ ⇒ 1
"Hi Greg, how are you" =~ m/G.*e/ ⇒ 1

* Means zero or more occurrences
Greedily chose longest match instead of: Gre

Literal meta-characters
in regular expressions

\ Prevents meta-meaning
"Hi Greg, how are you" =~ m/G\.// ⇒ undef
"Hi Greg, how are you" =~ m/G.*e/ ⇒ undef
**Regexp special characters**

- Quote the next metacharacter
- Match the beginning of the line
- Match the end of the line
- Match any character except a newline (\s modifier makes it also match a newline)
- Alternation
- Grouping
- Character class

**Regexp grouping**

```
my $line = "jill,bob,sam";
my ($first_part, $second_part) = ($1,$2);
print $first_part, $second_part;
```

This comma is the only literal character in the regular expression $1 $2

```
my ($first_part, $second_part) = ($1,$2);
my ($first_part, $second_part) = ($1,$2);
my ($first_part, $second_part) = ($1,$2);
print "Greedy" matching - the longest matching was chosen.
```

**Usefulness of regular expressions**

- Wrote 11,000 line static analysis tool for better understanding how C programmers used the C pre-processor in real programs
- Used regular expressions pervasively
- For example, to look for #if, #ifdef, or #endif preprocessor directives:

```
my /\s*#\s*(if(def)?| endif)\s.*$/;  # Greedy matching
```

**Subroutines**

```
sub comma_to_colon {
  my ($str) = (@_);
  $str =~ s/,/:/g;
  return $str;
}
$line = comma_to_colon($line);
$line  ⇒  "jill:bob:sam"
```

**References**

```
sub comma_to_colon {
  my ($ref_str) = (@_);
  $$ref_str =~ s/,/:/g;
}
comma_to_colon($$line);
$line  ⇒  "jill:bob:sam"
```

Extra $ to de-reference (like * in C/C++)

```
sub comma_to_colon {
  my ($str) = (@_);
  $str =~ s/,/:/g;
  return $str;
}
$line = comma_to_colon($line);
$line  ⇒  "jill:bob:sam"
```

Extra $ to de-reference (like * in C/C++)

```
sub comma_to_colon {
  my ($str) = (@_);
  $$ref_str =~ s/,/:/g;
}
```

creates a reference (like & in C/C++)

```
sub comma_to_colon {
  my ($ref_str) = (@_);
  $$ref_str =~ s/,/:/g;
}
```

Learning more...

- See my book recommendations online
- On-line links (see class web page)
- Perldoc, info pages, etc., e.g.:
  % perldoc CGI