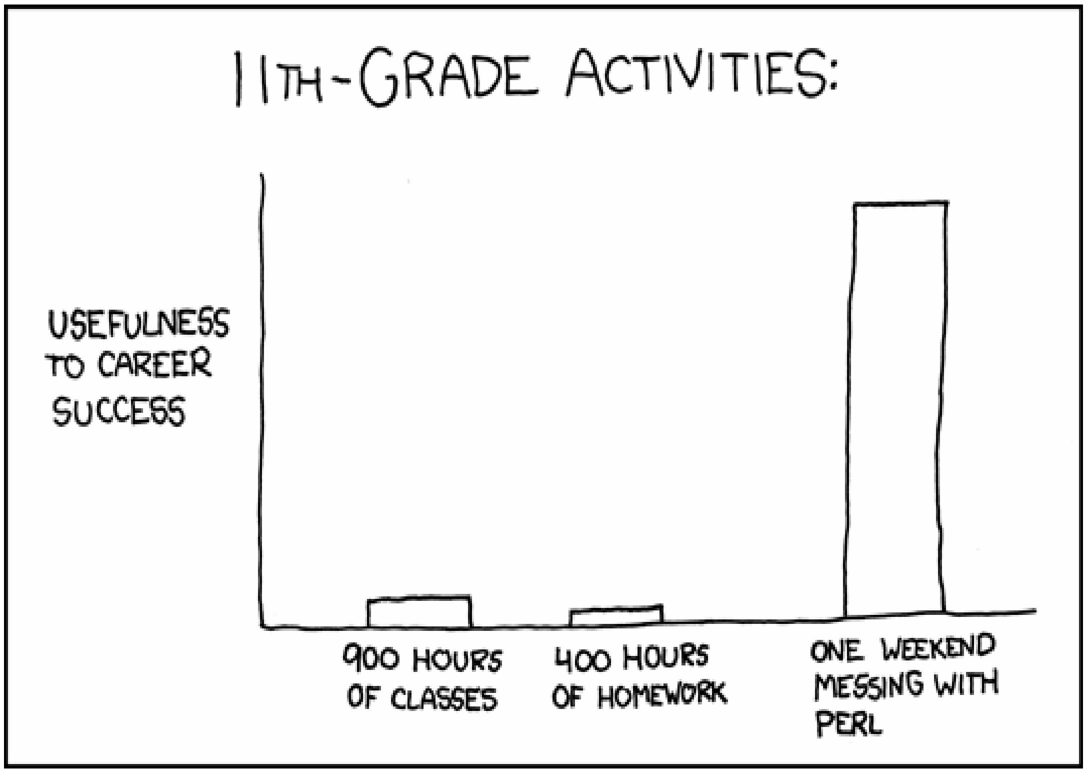
Perl

- **background**
- **basic language: variables, operators, expressions, control flow, …**
- **arrays and hashes**
- **scalar and list contexts**
- **file handles and I/O**
- **regular expressions and strings**
- **extension**

- **performance**
- **assessment**
- **Disclaimer: I am NOT a Perl expert**
- **see *Programming Perl, 3rd edition***
  Larry Wall, Tom Christiansen, Randal Schwartz (O'Reilly, 2000)
- **see www.cpan.org** (Comprehensive Perl Archive Network)

xkcd.com/519



Perl

- **developed ~1987 by Larry Wall**
- **a reaction to features lacking in Awk**
  - "Larry's first thought was "Let's use awk." Unfortunately, the awk of that day couldn't handle opening and closing of multiple files based on information in the files. Larry didn't want to have to code a special-purpose tool. As a result, a new language was born."
  - plus pieces from shell, sed, C, …
- **started small, now large & complicated**
  - "kitchen sink" language
  - we'll do only a very small part
- **system administration tool**
  - string processing
  - lots of functions to access (Unix) system calls
- **major scripting language for Web programming**
  - string processing
  - cgi-bin scripts for generating Web pages in response to queries

World's most boring example

```
for ($fahr = 0; $fahr <= 300; $fahr += 20) {
   printf("%3d %6.1f\n", $fahr, 5/9 * ($fahr-32));
}      # World's most boring example
```

- **while, for loops are like C**
- **if (…) {…} elsif (…) {…} else {…}**
  - {…} and terminating semicolons are required
- **scalar variable indicated by $: $name   ($ is required)**
- **arithmetic is float (5/9 is 0.555, not zero)**
- **variables hold strings or numbers as in Awk**
  - interpretation is determined by operators & context
- **operators:**
  - arithmetic operators much like C
  - string concatenation uses `.` ("dot")
  - relational operators are different for string comparison and numeric comparison: **`eq ne lt le gt ge` vs `==  !=  <  <=  >  >=`**
  - file test operators -f, -d, …
  - regular expression operators

Safety measures

- **Perl is often too forgiving**
  - like most scripting languages

- **-w** warns about potential errors like undefined or uninitialized variables
- **-T** ("taint") flag warns of potential problems from use of user-provided values in variables.
- **use strict** enforces variable declarations, etc.
- **my $var** declares variable
  - `my ($v1, $v2)` to declare several variables

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

use strict;
my $fahr;

for ($fahr = 0; $fahr <= 300; $fahr += 20) {
   printf("%3d %6.1f\n", $fahr, 5/9 * ($fahr - 32));
}
```

(Indexed) Arrays

- **array variable indicated by @:** `@arrname`

- **elements accessed as `$arrname[$index]`**

- **subscripts normally range from 0 to `$#arrname` inclusive**

- **echo command (two versions):**

```
for ($i = 0; $i <= $#ARGV; $i++) {
    if ($i < $#ARGV) {
        print "$ARGV[$i] ";
    } else {
        print "$ARGV[$i]\n";
    }
}

foreach $i (0 .. $#ARGV) {
     print $ARGV[$i]
        . ($i < $#ARGV ? " " : "\n");
}
```

Scalar and list contexts

- **two basic contexts: scalar and list**
  - an array is really a list

```
@arr = (1, 2.3, "hello", 45);
```

- **many operators take a list as argument**
  - and often return a list: keys, sort, reverse, grep, …

- **many operators can produce a scalar or a list, depending on context in which operator occurs**
  - sort LIST produces a list
  - print LIST produces a scalar (string)
  - print sort LIST produces a sorted scalar
  - split SCALAR returns a list/array

```
@wds = split " ", $_
($n, $v) = split;
```

  - grep PAT, LIST produces a list

```
@nonb_lines = grep /./, @all_lines;
```

  - print sort grep(PAT, LIST) produces a sorted string

"There's more than one way to do it"

- **echo command using list contexts and conversions**

```
foreach $i (@ARGV) {
   print $ARGV[$i] . ($i < $#ARGV ? " " : "\n");
}


print join(" ", @ARGV) . "\n";


print "@ARGV\n";
```

Hashes (== associative arrays)

- **hashes are a separate type, indicated by %: `%hashname`**
- **subscripts are arbitrary strings**
  - accessed as **`$hashname{$str}`**
- **example: add up values from name-value input**

pizza	200
beer	100
pizza	500
coke	50

```
my ($i, %val, @wds);
while (<>) {  # loop over ARGV files
    @wds = split;
    $val{$wds[0]} += $wds[1];
}
foreach $i (keys %val) { # note keys
    print "$i  $val{$i}\n";
}
```

- **AWK version:**

```
    { val[$1] += $2 }
END { for (i in val) print i, val[i] }
```

File handles and I/O

- **open function connects file to file handle**
  - **`open(FH, "file")`** for reading
  - **`open(FH, ">file")`** for writing, >> for append
  - **`open(FH, "| cmd")`** for piping to cmd
  - **`open(FH, "cmd |")`** for piping from cmd
  - STDIN, STDOUT, STDERR already open
- **copy input lines to output:**

```
while (<>) {    # all files in input list
   print "$_"; # $_ is input line with \n
}
```

- **add up names and values:**

```
open(SORT, "|sort +1 -nr");
while (<>) {
   ($n, $v) = split;
   $val{$n} += $v;
}
foreach $i (keys %val) {
   print SORT "$i\t$val{$i}\n";
}
```

- **close function breaks connection, recovers resources:**

```
close FH
```

ARGV and file handles

- **special cases for ARGV**
  - **@ARGV** is array/list of all command line arguments
  - **<ARGV>** is array/list of all lines of all arguments
  - **<>** is an abbreviation for **<ARGV>**
- **cat command**

```
foreach $i (@ARGV) {
   open(IN, $i) or die "can't open $i: $!";
   while (<IN>) {
      print $_;
   }
   close IN;
}

while (<ARGV>) { # each line of each file arg
   print "$_";
}

while (<>) {  # ARGV is implicit
   print;     # $_ is implicit
}

 print <ARGV>; # print gives scalar context

 print <>;     # ARGV is implicit
```

Regular expressions and pattern matching

- **m//   match operator**
  - **m/re/** matches (is true) if **re** matches operand
    `if (m/[yn]/) …`   implicit operand is $_
    `if (/[yn]/) …`   implicit operand is $_
- **s/re/repl/   substitution operator**
  - replace **re** with **repl** in target

- **tie these to an explicit string with =~ operator**
  `$str =~ s/re/repl/g;` # g = global, i = ignore case
  `if ($str =~ /[yn]/i) …`

- **shorthands**
  - \d = digit, \D = non-digit
  - \w = "word" character **[a-zA-Z0-9_]**, \W = non-word character
  - \s = whitespace, \S = non-whitespace
  - \b = word boundary, \B = non-boundary

- **substrings**
  - matched parts are saved for later use in $1, $2, ...
    `s/(\S+)\s+(\S+)/$2 $1/` swaps first two words

- **there's lots more!**

Control flow revisited

```
if (…) {…} elsif (…) {…} else {…}
  stmt if expr;
  stmt unless expr;

while (…) {…}
  stmt while expr;
  until (…) {…}

for ( ...; ...; ... ) {…}

foreach $i (list) {…}
foreach (list) { … $_ … }

sub name() {
   # array  @_  contains the arguments
}
```

- **elements are $_[0], $_[1], etc.**
  `my $x = shift @_` is like shell's shift cmd
  `my $x = shift      # default is @_`

Surprises, gotchas, etc.

- **@Arr define an array; $Arr[$i] references an element**
  - same for %hash, $hash{$i}
- **string comparisons: eq not ==, ne not !=, …**
- **no interpretation of \ inside '...'**
  - if ($c eq '\t') doesn't match a tab
- **elsif, not else if**
- **print ($i == $#ARGV) ? "\n" : " "**
  - needs either extra outside parentheses or no parentheses
- **$#Arr is the index of last elem, starting at 0**
  - not the number of elements; `$Arr[$#Arr]` is the last element
- **if (defined($x)) needed if use strict**
- **all lines in a single string:**
  - `my @in = <>; $str = join "", @in;`
- **chop returns char chopped, not resulting string**
  - **chomp** returns number of characters dropped!
- **foreach $i (%hash) is DIFFERENT from foreach $i (keys %hash)**

Review: Formatter in AWK

```
#!/bin/sh
# f - format text into 60-char lines

awk '
/./  { for (i = 1; i <= NF; i++) addword($i) }
/^$/ { printline(); print "" }
END  { printline() }

function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line space w
    space = " "
}

function printline() {
    if (length(line) > 0)
        print line
    line = space = ""
}
' "$@"
```

Formatter in Perl

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

my ($line, $space);
while (<>) {
    chomp;   # get rid of newline if it's there
    if (/^$/) {
        printline();
        print "\n";
    } else {
        foreach (split()) {
            printline() if (length($line)+length($_) > 60);
            $line .= $space . $_;
            $space = ' ';
        }
    }
}
printline();

sub printline() {
    print "$line\n" if (length($line) > 0);
    $line = $space = '';
}
```

What makes Perl successful?

- **rich language, regular expressions, strings**
  - with lots of support beyond bare minimum
  - object-oriented extensions
  - permits asynchrony, exceptions, etc.
- **access to underlying system (especially Unix) and to networks**
- **comparatively efficient**
  - fastest scripting language
- **enormous set of libraries**
  - Perl modules for almost anything
  - extensible by calling C or other languages
- **embeddings of major libraries in Perl**
  - e.g., Perl/Tk
- **large user community**
  - open source
- **standard: there is only one Perl**

Perl vs. Awk

- **most tradeoffs in Awk were made to keep it small and simple**
- **most tradeoffs in Perl were made to make it powerful and expressive**
- **domain of applicability**
  - Awk better for true 1-liners
  - Perl scales to big programs, does system applications much better
- **learning curve**
  - Awk is a lot simpler
- **efficiency**
  - Perl is definitely faster
- **standardization**
  - there's only one Perl (?)
- **program size, installation, environmental assumptions**
  - Perl is big, uses a big configuration script, takes advantage of the environment
  - Awk is small, uses no configuration script; does not try to adapt to environment