PERL

A language by Larry Wall

Practical Extraction and Report Language

or

Pathologically Eclectic Rubbish Lister
First, some terminology

- Commandline arguments: arguments you pass to a program, when you call it:
  
  ```
  $ ls -l -a
  here -l and -a are commandline arguments
  ```

- Standard input/output/error: is connected to the console i.e., the perl program
  
  ```
  print ‘Hello, World!’
  ```
  Writes “Hello, World!” to stdout, i.e., the console (terminal, screen)
PERL

- Developed by Larry Wall (late 80s) as awk replacement
- Very useful, since:
  - platform independent
  - Has powerful default libraries for many applications
    - Web/CGI, Databases, Sockets, ...
  - Powerful text processing (regular expressions et al.)
Perl (2.)

- Interpreted language with C-like syntax (with "integrated" awk, sed, and sh)
- Highly optimized for manipulation of printable text (but can also work with binary data)
- Useful for sysadmin jobs
- Rich enough for almost all programming tasks
- "A shell for C programmers" [Larry Wall]
Perl (3.)

- Some criteria for the design of perl:
  - There’s more than one way to do it
  - Make it simple to use natural language constructs (‘print it’)
  - Use meaningful defaults to reduce number of declarations
  - Don’t be afraid to use context as a syntactic tool
  - A huge language, where users will learn a subset

- How does this all work?
  - A language that make implementing useful systems easy
  - Readability of code can be a problem
Example

Example code

Don’t let this confuse you

```perl
while (<>) {
    next if /^#/;
    ($x, $y, $z) = /(/S+)/s+(/d\d\d)/s+(foo|bar)/;
    $x =~ tr/a-z/A-Z/;
    $seen{$x}++; 
    $z =~ s/foo/fear/ && $scared++; 
    printf "%s %08x %-10s\n", $z, $y, $x 
    if $seen{$x} > $y;
}
```
Using Perl

- The most basic program (ever)
  ```perl
  print "Hello, world\n";
  ```
- Can be executed with as a command line argument to perl:
  ```bash
  $ perl –e 'print "Hello, world\n";'
  Hello, world
  $'
- If the code is in a file `hello.pl`, then
  ```bash
  $ perl hello.pl
  Hello, world
  $`
Using perl (2)

- Alternatively you can add the following line to hello.pl
  ```perl
  #! /usr/bin/perl
  print „Hello, world\n“;
  ```

- Make the file executable
  ```bash
  $ chmod +x ./hello.pl
  ```

- And execute it with
  ```bash
  $ ./hello
  Hello, world
  $```
Syntactic Conventions

- Variables are prepended by a $ sign:
  \$x=1;

- As C/C++ programmer, you will often write:
  \x = 1; instead of \$x=1;

- Perl will answer with the following message:
  Can\'t modify constant item in scalar assignment ...

- The error message will always contain the line number (xyz)

- Solution: Got to line xyz and add the dollar sign
Syntactic Conventions (2)

- Two kinds of strings:
  - with interpolation ($x$ in the string is replaced by the value of $x$, more on next slide)
  - without interpolation

- Examples:
  ```php
  $answer = 42;  # An integer (or a float)
  $pet = "Camel";  # string
  $msg = "Ich liebe $pet";  # string with interpolation ($pet)
  $msg = "Gehe ${pet}s";  # string with interpolation ($pet)
  $cost = 'Es kostet $100';  # string without interpolation
  $dst = $src;  # Assignment
  $x = $y + 5;  # Expression
  $cwd = `pwd`;  # Assign output of the command `pwd`
  Note: backticks
  $stat = system("ls $d")  # Numeric status code of command `ls $d`
  Kommandos
  ```
Syntactic Conventions (3)

- Interpolated strings will:
  - interpolate the value of every variable in the string (e.g. \$x
  - Interpolate backslash escape sequences (e.g., \n \t \")

- Not interpolated strings will:
  - interpret $ \ \text{etc. as ordinary character}
  - interpret \` to allow embedded single quotes

- Examples:
  \$x = 1; \$y = "xyz"; \$z = 'abc';
  \$a = "a: why isn't $x better than "$y" or \$z";
  \$b = 'b: why isn't $x better than "$y"C or \$z';
# a: why isn't 1 better
# than "xyz" or $z$

# b: why isn't $x$ better\ than "$y" or \$z
Comparision Operators

- Perl uses different operators for strings and numbers.
  - You must specify with comparision you want!!!

<table>
<thead>
<tr>
<th>Operation</th>
<th>numeric</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>==</td>
<td>eq</td>
</tr>
<tr>
<td>not equal</td>
<td>!=</td>
<td>ne</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>lt</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>gt</td>
</tr>
<tr>
<td>less or equal</td>
<td>&lt;=</td>
<td>le</td>
</tr>
<tr>
<td>Comparision</td>
<td>&lt;=&gt;</td>
<td>cmp</td>
</tr>
</tbody>
</table>

$a <=> $b is 0 if they are equal, 1 if $a is greater, -1 if $b is greater.
Logical Operators

- Perl has two kinds of logical operators
  - C-like
  - english like
- The second kind has lower precedence
  - for use between statements
  - Hint: always use parentheses!!!

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>x &amp;&amp; y</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>! x</td>
</tr>
<tr>
<td>and</td>
<td>x and y</td>
</tr>
<tr>
<td>or</td>
<td>x or y</td>
</tr>
<tr>
<td>not</td>
<td>not x</td>
</tr>
</tbody>
</table>
Logical Operators (2.)

- Short circuit evaluation (like every other language)
- Example how you can use them in statements:
  ```
  if (! open (FILE, „myFile“)) {
      die „Can‘t open myFile“;
  }
  # can be replaced with
  open (FILE, „myFile“) or die „Can‘t open myFile“;
  ```
Variables

- Perl supports 3 kinds of variables / data structures
  - **Scalars** ... Atomic value (number or string)
  - **Arrays** ... List of values, indexed by numbers
  - **Hashes** ... Group of values, indexed by strings

- Variables don’t have to be declared or initialized
- If a variable is not-initialized it will have a value of 0 (or the empty string or an empty list)
- NOTE: Typos in variable names don’t yield a parse error
  - `print "abc=$ac\n";` instead of `print "abc=$abc\n";`
Variables (2)

- Many scalar operations have an idea of a default source/target
- If no argument is given, the special variable $_ is used!
- Advantage:
  - Often useful to write short programs
- Disadvantage:
  - Confusing for (new) users

(The use of $_ is similar to using ‘it’ in english)
Scalars

- Four kinds of datatypes: String, Integer, Float, Boolean
- Values of scalars are automatically casted and interpreted based on context (e.g., through the operator)

Examples:

```
$x = '123';  # The string „123“ is assigned to scalar $x
$y = "123 "; # The string „123“ is assigned to scalar $y
$z = 123;   # The numeric value 123 is assigned to scalar $z
$i = $x + 1; # $x is interpreted as integer
$j = $y + $z; # $y is interpreted as integer
$a = $x == $y; # compare $x, $y numerically, store in $a
$b = $x eq $y; # compare $x, $y as strings
$c = $x . $y; # concatenation of strings $x, $y
```
Arrays (Lists)

- An Array is a sequence of scalars, indexed via positions (0, 1, 2, ....)
- The whole array is accessed with @array
- Individual elements with: $array[index]
- $#array returns the index of the last element
- Examples:
  
  $a[0]="first string"; $a[1] = "2nd string"; $a[2] = 123;

Or

@a = ("first string", "2nd string", 123);
print "Index of the last element is $#a\n";
print "Number of elements is ", $#a+1, "\n";
Arrays (Lists) (2)

- Arrays don’t have to be declared or initialized
- Arrays grow and shrink dynamically
  - `@h = 99;` # create an array with 99 elements
- "Missing" elements are interpolated
  - `$abc[0] = "abc"; $abc[2] = "xyz";` # Accessing `$abc[1]` yields "" (the empty string)
- Assignments to/from complete arrays possible
  - `@numbers = (4, 12, 5, 7, 2, 9);` ($a, $b, $c, $d) = @numbers;
Arrays (Lists) (3)

- Arrays can be accessed element by element

```perl
@nums = (23, 95, 33, 42, 17, 87);
$sum = 0;  # optional
for ($i = 0; $i <= $#nums; $i++) {
    $sum += $nums[$i];
}
```

Or

```perl
foreach $x (@nums) {
    $sum += $x;
}
```
Arrays (Lists) (4)

- The operators `push` and `pop` work on the „right“ end of an array

  ```perl
  @a = (1, 3, 5); # (1, 3, 5)
push (@a, 7);   # (1, 3, 5, 7)
$x = pop @a;    # (1, 3, 5)
  ```

- Other useful operations on arrays:
  ```perl
  sort(@a)       # yields a sorted version of @a
  reverse(@a)   # yields the reversed version of @a
  shift(@a)     # like pop(@a), but on the left end
  unshift(@a,$x) # like push(@a,$x), but left end
  ```

- `push`, `pop`, `unshift`, `shift` can be used to implement stacks and queue
Digression: make Perl stricter

- Perl doesn’t require you to declare variables => typos not easily recognized
- You can “declare” variables
  ```perl
  my $var=1;     my @arr;     my ($x,$y,$z);
  ```
- These pre-declared variables have local scope (important in subroutines!!)
- Sugar: At the beginning of your script write:
  ```perl
  use strict;
  ```
  Now Perl requires you to declare all variables.
- Using `my` and `use strict` makes debugging MUCH easier
Control structures

- a **semicolon** must terminate each Perl statement, e.g.:
  
  ```
  $x = 1;
  print "Hello";
  ```

- **All** statements with control structures **must** be grouped by curly braces `{}`, e.g.:

  ```
  if ($x > 9999) {
    print "x is big\n";
  }
  ```

  No single line if-statements etc. without braces!!
Selections and if

- Done using if ... elsif ... else

```java
if ( boolExpr1 ) {
    Statements 1;
} elsif ( boolExpr2 ) {
    Statements 2;
} ...
else {
    Statements n;
}
```

- There’s no switch/case
Selections and if (2)

- if can also be used as operator: the statement

```perl
if ( $x < 0 ) {
    print "X is negative";
}
```

can be written as

```perl
print "X is negative" if ($x < 0);
```

or as

```perl
print "X ist negative" unless ($x >= 0);
```
Iteration

- while, until, for, foreach

```plaintext
while ( boolExpr ) {
  statements;
}
until ( boolExpr ) {
  statements;
}
for ( init; boolExpr; step ) {
  statements;
}
foreach var ( list ) {
  statements;
}
```
Iteration (2)

Example: Calculate

\[ pos = \frac{1}{n} \]

# Methode 1: while

```php
$pow = $i = 1;
while ( $i <= $n ) {
    $pow *= $k;
    $i++;
}
```

# Methode 2: for

```php
$pow = 1;
for ( $i = 1; $i <= $n ; $i++ ) {
    $pow *= $k;
}
```

# Methode 3: foreach

```php
$pow = 1;
foreach $i ( 1 .. $n ) {
    $pow *= $k;
}
```

# Methode 4: Operator

```php
$pow = $k ** $n;
```
Iteration (3)

- **foreach** uses `$_` if no variable given
  ```perl
  @countdown = (10,9,8,7,6,5,4,3,2,1);
  foreach (@countdown) {
    print;  # uses $_
    print "\n";
  }
  ```
- or even
  ```perl
  foreach (10,9,8,7,6,5,4,3,2,1) {print; print "\n";};
  ```
- Or
  ```perl
  foreach (10,9,8,7,6,5,4,3,2,1) {print "$_\n";};
  ```
Input / Output

- Files are accessed via handles
- The expression `<Handle>` for an input filehandle means „read the next line of this file“
  
  e.g.: $line = `<STDIN>`;
  
  ... save the next line of standard input into variable $line.
- Output handles are used as first argument to print:
  
  e.g.: print REPORT "Report for $today\n";
- ... writes a line into the file associated with handle REPORT
**Input / Output (2)**

- Example (a simple cat):

  ```perl
  #!/usr/bin/perl
  # Copy stdin to stdout
  while ($line = <STDIN>) {
    print $line;
  }
  ```

- Or simpler

  ```perl
  while (<STDIN>) { print; }
  ```

- Or even

  ```perl
  print <>;
  ```
Input / Output (3)

- Handles are associated with a file by the `open` command:
  - `open(DATA, "< data");` # read from file "data"
  - `open(RES, "> result");` # write to file "result"
  - `open(XTRA, ">> stuff");` # append to file file "stuff"

- Handles can also be associated with pipelines to read/write from Unix commands:
  - `open(DATE, "/bin/date |");` # read output from date program
  - `open(FEED, "| more");` # send output to more program

- Opening a handle can fail: error handling:
  - `open(DATA, "< data");` or `die "Can't open data file";`

- Handles are closed by calling `close(HANDLE)`
Input / Output (4)

- The special file handle `<>`
  - Treats command line arguments as file names
  - Opens and reads all of them
- If there are no command line arguments:
  - `<>` represents `<STDIN>`
- I.e., `<>` has the semantic that many Unix tools use
- Example:
  - `perl –e ‘print <>; ‘ a b c`
  - Prints the contents of files a, b, and c to stdout
String Functions

- Remove newlines (\n): chomp
- Example:

```perl
chomp($host = 'hostname');

while (<STDIN>) {
    chomp;
    ...
}
```
**Associative arrays (hashes)**

- **Hash**: Arrays indexed by strings
- A hash is a dictionary data structure with (key, value) pairs
- Access to the whole hash is done by using `%hashName`, e.g.:

  ```perl
  %days = ('Sun'  =>  'Sunday',
          'Mon'  =>  'Monday',
          'Tue'  =>  'Tuesday',
          'Wed'  =>  'Wednesday',
          'Thu'  =>  'Thursday',
          'Fri'  =>  'Friday',
          'Sat'  =>  'Saturday');
  ```
Associative arrays (hashes) (2)

- Individual elements are accessed with 
  $hashName{keyString}$

- Example:
  
  ```perl
  $days{“Sun“};       # yields „Sunday“
  $days{“Fri“};       # yields „Friday“
  $days{“dog“};       # yields „“ (empty string)
  $days{0};           # yields „“ (empty string)
  # Inserting a new element:
  $days{“dog“} = “Dog Day Afternoon“;
  # Replace the value for key „Sun“:
  $days{“Sun“} = “Soonday“;
  ```
Associative arrays (hashes) (3)

- To access the (key, value) pairs do:
  foreach $key (keys %myHash) {
    print "($key, $myHashs{$key})\n";
  }

- Or, if you just want the values without the keys:
  foreach $val (values %myHash) {
    print "$val\n";
  }
Associative arrays (hashes) (4)

Example (Collecting grades for per student)

- The input file should consist of (name, grade) pairs, separated by a space, one entry per line
- The output should be in the form (name, list of grades), where the grades are separated by commas

```perl
while (<>)
{
    chomp;
    ($name, $mark) = split;
    $marks{$name} .= '$mark,‘;
}
foreach $name (keys %marks)
{
    print "$name $marks{$name} ‘‘;
}
```
Associative arrays (hashes) (5)

- The `delete` command removes an entry (or entries) from hashes

To remove one entry:

```perl
delete $days{"Mon"};  # I don’t like Monday
```

To remove several pairs:

```perl
delete $days{"Sat", "Sun"};  # no weekend
```

Or the whole hash:

```perl
undef %days;
```
Perl Regular Expressions

- Since perl is based around string processing, regular expressions are an important part of the language.
- They can be used to:
  - To test whether a string matches a given pattern:
    ```
    if ($name =~ /[0-9]/) {print "name contains digit\n";}
    ```
  - In assignments to convert / replace parts of a string (e.g., to replace Mc with Mac in $name):
    ```
    $name =~ s/Mc/Mac/;
    ```
Perl Regular Expressions (2)

- A regular expression is a pattern of characters
- The simplest pattern is an ordinary character. It matches itself
- Patterns can be composed from other patterns
  - `ab` finds combinations `ab`
  - `ab|yz` finds `ab` or `yz`
  - `[0123456789]` finds a digit
  - `[0-9]` shorthand for the above
every character in range
  - `[range]`
  - `. ` matches any character (except \n)
  - `^` finds the start of the string
  - `$` finds the end of the string
  - `\` Escape for the next character
Perl Regular Expressions (2)

- **Quantifiers:**
  - `x*` 0 or more occurrences of `x`
  - `x+` 1 or more occurrences of `x`
  - `x?` 0 or 1 occurrences of `x`
  - `x{n,m}` between `n` and `m` occurrences of `x`

- Use parentheses to group patterns for quantifiers
  - `(abc)*` 0 or more occurrences of `abc`

- Perl also knows some short cuts:
  - `\d` finds a digit, i.e., [0-9]
  - `\D` finds everything that isn’t a digit, i.e., [^0-9]
  - `\w` finds „word“ characters, d.h. [a-zA-Z\_0-9]
  - `\s` finds a whitespace, i.e., [ \t\n\r\f]
Perl Regular Expressions (4)

- The default matching semantic is
  - match the first possible occurrence
  - then use the longest possible match
- Example: matching `/ab+/` against `abbabbb`:
  - finds `abbabbb`
  - not `abbabbb` (because `abb` is longer)
  - not `abbabbb` (because it is not the first match)
Perl Regular Expressions (5)

How can we use them:

- Just matching:
  m/pattern/[options] or /pattern/[options]
  e.g., $string =~ m/pattern/ ... yields a boolean

- Match with replace (substitute):
  s/pattern/replacement/[options]
  e.g., $string =~ s/Mc/Mac/ ... replaces the first occurrence of Mc with Mac in string.

Some options:

/i   Ignore case (case-insensitive)
/g   global match: match (resp. replace) all occurrences
/m   Multi line pattern: ^,$ match start and end of line (instead of string)
Perl Regular Expressions (6)
Accessing found patterns

- Use parentheses ():
  - $nn: The $nnth parentheses expression
  - $MATCH or $&: The complete part of the string that matched

- Example:
  ```perl
  if ($name =~ /[vV]on\s+(.*)/) {
      print "$1 hatte adelige Vorfahren\n";
  }
  ```
Lists and strings

- Often we have strings with a list of elements, e.g., “1,6,42,34”

- How can we transform this into a real list: use \texttt{split}
  
  - \texttt{split(/pattern/, $string)} returns an array (list), e.g.:
    
    ($a,$b,$c,$d) = split /,/, $string \\
    or \\
    @x = split /,/, $string
  
  - If we specify a list (e.g. ($a,$b,$c)) with not enough elements, then the last element $c$ will receive the remainder (here $c = (42,34)$)

- \texttt{join} is the reverse operation
  
  - \texttt{join(‘char‘, @array)} returns a string
Lists and strings (2)

Examples:

```perl
$marks = "99,67,85,48,77,84";
@listOfMarks = split /,/, $marks; # assigns (99,67,85,48,77,84) to @listsOfMarks

$sum = 0;
foreach $m (@listOfMarks) {
    $sum += $m;
}

$newMarks = join ':', @listOfMarks; # assigns the string “99:67:85:48:77:84” to $newMarks
```
Special Variables

- Perl defines some special variables with information about the execution environment
- They look like:
  - $! $@ $# $$ $% ...
- $_ is very important:
  - Default to assign results
  - Default argument for many operations
- Good use: „nice, short“ programs
- Bad use: „cryptic“ programs
**Special Variables (2)**

$ _ $ Default variable

$0 $ Filename of the currently running script (program)

$1 $ 1st () match of previous regular expression

$2 $ 2nd () of previous regular expression

.....

$$ $ Process-Id of currently running Perl script

@ARGV $ List of command line arguments

%ENV $ Hash with environment variables
Special Variables (3)

Example (a simple echo):

```perl
for ($i = 0; $i <= $#ARGV; $i++) {
    print "ARGV[$i] \n; }
print \n;
or
foreach (@ARGV) {
    print "$_ \n; }
print \n;
```
Function Calls

- Notation for function calls in Perl:
  
  ```perl
  &func (arg1, arg2, ..., argn);
  
or
  @args = (arg1, arg2, ..., argn);
  &func(@args);
  ```

- In almost all cases one can omit the & operator:
  ```perl
  func (arg1, arg2, ..., argn);
  ```

- In almost all cases one can omit the parentheses:
  ```perl
  func arg1, arg2, ..., argn;
  ```

- Parameters are received by the subroutine as a special array `@_`. Its values can be copied to local variables.
Functions (Subroutines)

Example:
```perl
$result = &simple($alpha, $beta, $gamma);
sub simple {
   my ($x, $y, $z) = @_;  
   my ($sum, %seen);  
   return $sum;
}
```

Or:  `$result = &simple(@list)`

If `return` is omitted: return value is last evaluated expression

Passing arrays and hashes to subroutines is tricky (need references)
File test operators

- Perl has a number of operators to test files
- Similar to test command in Unix
- -r, -w, -x test whether a file is readable, writeable, executable respectively

Example:

- r "dataFile" && open DATA, "<dataFile";
# or with an if statement:
if ( -r "dataFile") {
    open DATA, "<dataFile";
}


Syntactic Conventions (short summary)

<table>
<thead>
<tr>
<th>CharType</th>
<th>Example / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>comment</td>
</tr>
<tr>
<td>$</td>
<td>scalar</td>
</tr>
<tr>
<td>@</td>
<td>array</td>
</tr>
<tr>
<td>%</td>
<td>hash</td>
</tr>
<tr>
<td>&amp;</td>
<td>subroutine</td>
</tr>
<tr>
<td>none</td>
<td>handle</td>
</tr>
</tbody>
</table>

- **#** comment: remainder of line is comment
- **$** scalar: Variables with atomic value
- **@** array: list of values, indexed by integers
- **%** hash: (key, value) pairs with arbitrary scalar as key
- **&** subroutine: Calls (or declares) a subroutine
- **none** handle: File handle (used to read/write files)

- **Other constructs:**
  - `<HANDLE>`: read line from HANDLE
  - `my`: declare a variable (scalar, array or hash)
Literature

❖ For starters
  ❖ Programming Perl
    Larry Wall, Tom Christiansen, Jon Orwant
  ❖ Learning Perl
    Randal L. Schwartz, Tom Christiansen
  ❖ A Little Book on Perl
    Robert W. Sebesta
Literature (2)

- Advanced
  - Perl Cookbook
    Tom Christiansen, Nathan Torkington, Larry Wall
  - Mastering Algorithms With Perl
    Jon Orwant, Jarkko Hietaniemi, John MacDonald, John Orwant
  - www.perl.com
    von verschiedenen Autoren, on-line Aufbewahrungsort für Perl Informationen
  - www.cpan.org
    von verschiedenen Autoren, übergreifendes Perl Archiv Netzwerk
Online documentation

- Central documentation piece: perldoc program
  - $ perldoc -f FUNCTION .... documentation for FUNCTION
  - $ perldoc -q SEARCHTERM ... look in the FAQ for SEARCHTERM
  - $ perldoc perlre ... Regular expression documentation (also try perlretut, perlrequick)

- or in the Web: http://perldoc.perl.org

- Beta Versions, Perl Packages, additional libraries
  http://www.cpan.org