PERL

by

Erick Antezana San Román

<erant@psb.ugent.be>

CONTENTS

- Introduction
- Scalar data
- Arrays and lists
- Structures of control
- Hashes
- I/O
- Regular expressions

CONTENTS

- Functions
- Diverse structures of control
- File handles
- Formats
- Access to directories
- Files and directories manipulation
- Process management

CONTENTS

- Data transformation
- Access to databases from the system
- Databases

INTRODUCTION

- PERL history:
  - **P**ractical **E**xtraction and **R**eport **L**anguage
  - Larry Wall
  - System reports: **awk** was too “small”
  - Grew up parallel to UNIX
  - Lots of features such as portability

- Perl purpose
  - Texts manipulation
  - Files and processes
  - WWW
- Distribution
  - free (GNU)
  - http://www.perl.com/CPAN
  - Amiga, Atari ST, Mac family, VMS, OS/2, MS-DOS, Windows NT, Windows 9X , Windows XP and UNIX (and their dialects)

- Basic concepts
  - **shell script** = sequence of shell commands embedded in a text file
  - **script perl** = perl program
  - There is no **"main"** like in C
  - Comments till the end of the line: #
  - The **perl** interpreter does a **parsing** and **compiles** the program before executing it
  - **compiled**: parsing before executing
  - **interpreted**: there is no object code generated

- An overview of **perl**
  - The “classic” program “hello world”

```
#!/usr/local/bin/perl -w
print (“hello world\n”);
```

  - Example 2

```
#!/usr/local/bin/perl -w
print “What is your name? ” ;
$name = <STDIN>;
chomp ($name);
print “Hello $name!!!\n”;
```

SCALAR DATA

- What is an scalar data?
  - It is the simplest type used in **perl**
  - It is a number or string
- Numbers
  - All the numbers (integers, float) use the same internal format = all of them are floating point values with double precision

- Examples:
  - **float**: -1.234e-5
  - **integer**: 1024
  - **octal**: 0377 # 255 in decimal
  - **hexadecimal**: 0xff # 0377 in octal
- Strings
  - Sequence of characters without limits
  - Standard set of characters
  - Two types: **single-quoted** and **double-quoted**

- Single-quoted = enclosed within apostrophes

  'perl' # four characters: p, e, r, l

  'PERL\n' # PERL followed by a backslash and **n**

- Double-quoted = within double quotes ;-)
  - **"perl\n":** perl followed by a new line
  - **"I\tdo not understand":** I, tab, do not understand
    - **\n** **new line**
    - **\t** **tab**
    - **\b** **backspace**
    - **\cC** **CTRL-C**
    - **** **backslash**
    - **\007** **octal 007 = bell**

- Operators = superset from Pascal, C, Java
- Operators for numbers

```
5 + 6           # 5 plus 6
1.2 - 3.4       # 1.2 minus 3.4
5 * 6           # 5 times 6
3.14 / 2.1      # 3.14 divided by 2.1
2**3            # 2 power 3 (FORTRAN)
10 % 3                  # 10 modulus 3
<, <=, ==, >=, > y  !=      # logic comparison
6 > 1           # returns TRUE
10 != 10        # FALSO
```

- String operators
  - **“exam” . “ple”** # example, concatenation
  - **eq, ne, lt, gt, le and ge**
  - **“hello” eq “perl”** # FALSO
  - **7 < 30**
  - **7 gt 30**
  - **“perl” x 3** # is “perlperlperl” (repetition)
- Precedence

  *** / % x** more precedence than **+ - .**

  ! more than **&&** more than ||

- Examples:

  30 / 6 * 3 # left associativity= 15

  72 / 12 / 3 # (72/12) /3 = 6/3 = 2

- Conversion between number and string
  - depends in the context

    "perl". (4*6) # produces "perl24"

    " 1.23perl" + 2 # produces 3.23

- Scalar variables
  - Holds scalar data
  - perl is case sensitive
  - Must begin with **$** (name limited to 255)

- Operators and Functions

```
$age = 30;
$birthday = 2006 - $age;
$b = 3 + ($v = 3);  # $b = 6
$b += 6;            # $b = 12
$v = ($b += 6);     # $v and $b have 18
$b = 2;
$v = ++$b;          # $v = 3 and $b = 3
$w =$b++;           # $w = 3 and $b = 4
$name = “erick”;
$k=chop($name);  # $name = “eric” y $k = “k”
$in = “hello world\n”
chomp($in);   # $in has no \n anymore
```

- Interpolating scalars within strings

  $nombre = “\uerick”;

  $info = “His name is $nombre”; # $info = “Erick”

  $info2 = “His name is \$nombre”; # no interpolation

- <STDIN> as scalar value

  Chomp ($name = <STDIN>);

- Output using **print**

  print “Hello my name is $name\n”;

- The value **undef**
  - Default value
  - **undef:** 0 in numbers and ‘’ (empty string) in strings

ARRAYS and LISTS

- What is a list and what is an array?
  - **List** = ordered scalar data
  - **Array** = variable holding a list
- Literal representation

  (1, 2, 3, 4) # list with: 1, 2, 3 y 4

  ( ) # empty list

  (1 .. 5, 6, 66, 666) # list with 1,2,3,4,5,6,66,666

  (3.3 .. 7.2) # same as (3.3, 43, 5.3, 6.3)

- The function “quote word”

  @arr = (“hugo”, “paco”, “luis”);

  @arr = qw(hugo paco luis); # the same
- Variables
  – an array stores a list (use **@**)
- Operators and functions

  @costs = (2, 3, 8);

  @prices = @costs;

  @one = 1; # @one = (1);

  @long = (1, @costs, 10);

  ($a, $b) = (2, 3);

- More examples:

($a, @a) = (0, 1, 2, 3); # $a = 0 and @a=(1,2,3)

($a, $b) = ($b, $a); # swap

$a = @a; # $a has the size of @a = 3

($a) = @a; # $a has the first element of @a

@a = @b = (1, 2);

$a[0] = 2; # now @a = (2, 2)

@a[0,1] = (1,0); # same as ($a[0], $a[1])=(1,0)

($c) = (9, 8)[1]; # $c = 8

@a = @a[@a]; # @a = (0,1)

$a[5] = 0; # @a = (0,1,undef,undef, undef,5);

- Last index

  @a = (1,2,3,4);

  print $#a; # prints 3 (last index)

  print $a[-2]; # prints 3, from backwards

- push and pop (use an array like a stack)

  push(@arreglo, $nuevo);

  $viejo = pop(@arreglo); # removes $nuevo

- shift and unshift (operations on the left)

  unshift(@arr, $primero); # insert at the beginning

  $x = shift(@arr); # get the first one

- Some functions

  @a = (1,2,3,16,32);

  @b = reverse(@a); # @b = (32,16,3,2,1);

  @s = sort(@a); # @s = (1,16,2,3,32);

  @c = (“hello\n”, “how\n”, “are you?”);

  chomp(@c); # @c=(“hello”, “how”, “are you?”)

- List and scalar context
  - **Law: “**if an argument is expected to be **scalar/list**, we say that it is being evaluated in **scalar/list** context**”**

- <STDIN> as an array

  @arr = <STDIN>; # read in list context

- Array interpolation

  print "just read: @arr";

  # will show all the read lines from the standard input

- Example

  – Order an entered list:

  print "Give your list: \n";

  print sort <STDIN>;

CONTROL STRUCTURES

- Sentences block
  - It's a sequence of sentences enclosed in curly braces.
- **if/unless**
  - **0** y **""** are false.

  if /unless (expression_one) { sentences_1 }

  elseif (expression_two) {sentences_2 }

  else { sentences_3 }

- **while/until**

  while/until (expression)

  { sentences }
- **do{}while/until**

  do{

  sentences

  } while/until expression;
- **for**

  for (initial_expr; test_expr; re-init_expr)

  { sentences }

- **foreach**

  foreach $i (@list)

  { sentences }

- Using **$_**

  @list = (2, 4, 6, 8, 10);

  foreach (reverse @list)

  { print; } # shows: 10,8,6,4,2

  foreach $par (@list)

  { $par *= 2; } # now @list = (4, 8, 12, 16, 20)

HASHES

- What is a hash?
  - Array of scalars with arbitrary indexes (keys=string)
  - The elements do not have a particular order
  - PERL holds the key-value pairs in an internal order facilitating their look up
- Hash variables
  - name (first character = letter) followed by **%**

- Example

  $numbers{“one”} = “uno’;

  $numbers{“two”} = “dos”;

  $numbers{“six”} = “seis”;

  $h{2.3} = “two point three”; # key = “2.3”

- Literal representation of a hash
  - Actually, there is no literal representation

  @test=%numeros; # @test = (“one”,“uno”, “two”, “dos”, “six”, “seis”);

  %numbers = %numeros; # copy

  %age=(“hugo”, 5,“paco”,4,“luis”,$numeros{“six”});

- Hash functions

  @claves=keys(%numbers);# @claves can be
  # (“one”,“two”,“six”)

  @valores= values(%numbers); # @valores can
  # have (“uno”, “dos”, “seis”)

  while (($k, $v) = each(%numbers)) # pairs

  { print “using $k, we find: $v\n”;}

  delete $numbers{“six”}; # deletes the pair six-seis

- Slices (other method to work with a hash)

  $iq{“pinky”, “the brain”} = (20, 250);

INPUT/OUTPUT

- Input using <STDIN>

  $line=<STDIN>; #evaluation scalar context

  @lines = <STDIN>; # list context

- Example

```
while (<STDIN>){  # similar to $_=<STDIN>
    chomp;        # similar to chomp($_);
    # some operations with $_
}
```

- Input using the diamond operator: **<>**
  - similar to **<STDIN>,** but **<>** extracts data from the files given in the command line

```
while (<>){
    print $_;
}
```

usage:

/home/>test file1 file2 file3

  - Actually **<>** works with the array **@ARG**, we can force the reading of other files…

- Output to STDOUT
  - perl uses **print** and **printf**
  - **print** returns **true(1)/false(0)** depending on what it did (success in printing=**true(1)**)
  - **printf** = formatted output similar to the one in C
  - example:

printf "%15s %5d %10.2f\n", $s, $n, $r;

  - prints **$s** in a place of **15** characters, then a place, then **$n** as a decimal integer in a place of 5 characters, then **$r** as a floating point with **2** decimals in **10** places and finally **\n**

REGULAR EXPRESSIONS

- What is a regular expression?
  - It's a **pattern** (template) to be matched with a string, similar to the ones used by: **grep**, **sed**, **awk**, **ed, vi**, **emacs** and some **shells**
- Example:

```
grep lucas /etc/passwd > result
while (<>){   # the same using perl
if (/lucas/){print $_;}}
```

- Patterns
  - point **“.”** matches a character (except **\n**)
  - a “class of characters” has the form: **[cars]**
  - **[aeiou]** matches a string having any of the vowels
  - **[a-zA-Z0-9]** any letter or digit
  - **[^0-9]** anything that is **NOT** a digit **(**negated: **^)**

Construction	Equivalent class	Negated construction
\d (a digit)	[0-9]	\D
\w (word char)	[a-zA-Z0-9_]	\W
\s (space char)	[\r\t\n\f]	\S

- More cases

  + : one or more previous characters or class

  * : zero or more previous characters or class

  ? : zero or one previous character or class

  /ab+c*de?/ matches: $_=“holaabbbbdmundo”

  /b{2, 4}/ matches a string with 2 to 4 b’s

  /b{2, }/ matches a string with >= than 2 b’s

  /b{0,3}/ matches 3 b’s maximum

  /b{2}/ matches 2 b’s exactly

  /u(.)cb(.)c\2ba\1/ matches : uxcbycybax ; 1, 2 make reference to the parentheses respectively

- More examples

  /day|evening|night/ only one of the three options

  \b, ^, $: to limit the patterns

  /fred\b/ will match: **fred** but not **frederick**

  /\bmo/ will match: **mo** or **mole** but not **ammo**

  ^z will match all line starting with a **z**

  a$ will match all line ending with an **a**

- precedence (high ==> low)

parenthesis:	( ) (?:)
multiplicities:	? + * {m,n}
sequences and delimitators:	abc ^ $
alternation:	|

- Operator =~

  $a = “hola mundo”;

  $a =~ /^ho/;   # true

- Ignoring case

  /paulatino/i;

- Using another delimiter

  /^\/**export**\/**home**/ # standard delimiter: **slash**

  m@^/export/home@ # delimiter: **@**

- Interpolation

  $name = “lucas”; $sentencia = “lucas es un pato”;

  $sentencia =~ /$name/;   # true

- Substitutions

  $_ = “carla y carlos”;

  s/carl/ell/g; # $_ has “ella y ellos”

  $dato =~ s/day/night/i; # ignore case

- Functions split and join

  $m=“fulanito::1:100:Test:/home/fulanito:/bin/sh”

  @record = split(/:/, $m);

  @datos=split; # same as @datos=split(/\s+/, $_);

  $cadena = join(“:”, @record);

FUNCTIONS

- Definition of a function

```
sub greeting{
    print “hello!!\n”;
}
```

  - defined in any place of the program
  - It’s possible to use global variables from within a sub
- Invocation

```
greeting();
```

- Returned values
  - is the value of the **return** sentence or the one from the last evaluated expression within the subroutine.
  - A **sub** can return a list when it is evaluated in a list context

```
sub list_a_b{
    return ($a, $b);
}
$a = 1;  $b = 2;
@c  = lista_a_b();  # @c has (1, 2)
```

- Arguments
  - Using the private and especial variable: **@_**

```
sub greeting{
    print "$_[0], $_[1]\n";  # first parameter
}
greeting("hello", "lucas");
```

- Private variables
  - Use the operator **my**

```
my($mi_variable) = 0; # initial value to a local var.
my($n, @arr) = @_;# initialize with parameters
```

- Semi-private variables (use **local**)
  - Variables declared with **local** are visible inside the functions called from the block within which they are declared.
  - It's possible to declare: **local ($_);** or any other special variable (not possible using **my**), that ensures that the value held by the special variable will be kept by the called function.
- Using **strict** (pragma)
  - **strict** asks that **ALL** variables be declared suing **my** before using them. Advantages: detect errors, …

MISCELLANEOUS CONTROL STRUCTURES

- The sentence **last** is used to get out of loops

```
while (something) {
    something_1;
    if (condition) {
            something_2;
            last;   # end up the loop!
    }
} # last "comes" here
```

- The **next** sentence with the loops

```
while (something) {
    do_something;
    if (some_condition) {
            do_something_more;
            next;   # alter the execution
    }
    other_stuff;
    # the next comes here
}
```

- **redo** sentence

```
while (something) {
    do_something;
    if (some_condition) {
            something_more_1;
            redo;   # go to the beginning of the loop
    }
    something_more_2;
}
```

- Labeled blocks

```
CYCLE: while (something) {
                    something_1;
                    if (condition) {
                          something_2;
                         last CYCLE; # leave the CYCLE
                    }
}
```

- Expression modifiers

```
expression if control_expression;
exp2 unless exp1; same as: unless (exp1) {exp2;}
```

exp2 while exp1; same as: **while(exp1) {exp2;}**

exp2 until exp1; same as: **until (exp1) {exp2;}**

- Example:

  chomp ($n = <STDIN>);

  $i =1; # initial value

  $i *= 2 until $i > $n; # iterate till finding >

- **&&** and || as control structures

  this && that; # same as **if (this){ that; }**

  this || that;# same as **unless (this) {that;}**

FILEHANDLES

- What is a filehandle?

  It's the name for an **I/O** connection between your **PERL** process and the rest of the world. **STDIN** is the connection between the **PERL** process and the standard **UNIX** input. Others: **STDOUT**, **STDERR**.

- Opening and closing a filehandle

  STDXXX: automatically opened by the **shell**

  open (FILEHANDLE, "file_name");

open (OUT, “>outfile”); # writing

open (APP, “>>logfile”);

close (APP);

- Reopening a **filehandle** and quitting the program closes automatically the previous filehandle.
- Try to close it: checking access problems, full disks, errors in remote servers, inaccessibility, and so forth.

- Function: **die**

  open (FH, “>>f”) || die “Errors while appending\n”;

- Using filehandles

```
open (PW, “>>/etc/passwd”) || die “error: $!”;
while (<PW>){
        chomp;
        print “entry: $_\n”;
  }
print PW “test:x:1:100:test:/home/test:/bin/sh\n”;
print STDOUT “hello!”;  # STDOUT is optional
```

- Testing files

```
print “file: ”;
chomp ($file = <STDIN>);
if (-r $file && -w $file) {
    # file exists and it’s possible to read it and write on it
}
foreach (@file_list)  {
   print “$_ is executable” if -x;  # same as -x $_
}
```

FORMATS

- What is a format?
  - It is a template for writing reports. It has two parts: **constant** (headers, labels, text) and **variable** (data to be retrieved)
- Format definition

format NAME =
fieldline
first_value, second_value
fieldline
first_value, second_value
.

- Format invocation (write)

```
format ADDRESSLABEL =
++++++++++++++++++++++
| @<<<<<<<<<<<<<<<<<< |
$name
| @<<<<<<<<<<<<<<<<<< |
$email
| @<<<<<<<<<, @<<<<   |
$telephone       $zip
++++++++++++++++++++++
.
```

- Cont…

```
open (ADDRESSLABEL, “>addresses”);
open (ADDRESS, “source”);
while (<ADDRESS>) {
    chomp;
    ($name,$email,$telephone,$zip) = split(/:/);
    write (ADDRESSLABEL);
}
```

- Source file:

```
john:john@perl.com:64563453:2345
peter:peter@perl.com:23423677:7547
```

- Fieldholders
  - **Text fields**

  @<<< left justification (4 places)

  @>>> right justification (4 places)

  @||| centered text
  - **Numeric fields**

  @#####.### the point is optional
  - **multi-line fields**

  @* used when the text has \n's
  - Full field

  ^<<<<< for a lot of justified text

- Header format (**_TOP**)

  $% variable defining the number of times the head format has been called

  format ADDRESSLABEL_TOP =

  My address : -- Page @<

  $%

  .

- By default the page is 60 lines long

DIRECTORY ACCESS

- Use the sytem call ***chdir***
  - **chdir (“/etc”) || die “not possible to go to etc”;**
- Directory handles:

  readdir

  opendir

  closedir

FILES AND DIRECTORY MANAGEMENT

- Delete a file: **ulink(“passwd.bk”);**
- Rename a file: **rename ($old, $new);**
- Hard links: **link(“old”, “new”);**
- Soft links: **symlink(“old”, “new”);**
- Create dir: **mkdir(“mi_dir”, 0777);**
- Delete dir: **rmdir(“mi_dir”);**
- Modif. permission: **chmod(0666,”file1”, .., “fileN”);**
- Modif. owner: **chmod($id,$gid,”file1”,...);**

PROCESSES

- **system: system “gcc –o test -DHARD”**
- **Back quotes:**

```
$now = “the date is ”.`date`; # scalar context
foreach $_ (`who`){
        print $_;
}
```

- **Filehandles as processes**

Ex1:

```
open (WHOPROC, “who|”); # for reading
@whosaid = <WHOPROC>; # acquisition
```

Ex2:

```
# for writing:
open (LPR, "| lpr -Pslatewriter");
# send it to the printer:
print LPR @my_report;
# close the handle
close (LPR);
```

- **Fork**
  - **similar to the UNIX primitive, creates a clone of the current process,**
  - **use: fork(), exec("proc"), wait(), waitpid ($pid) and exit.**

DATA TRANSFORMATION

- **Finding a string**

  $pos = index ($my_string, $my_pattern); # from left

  # $pos = 0 is the first position

  # $pos = -1 if it does not exist

  $pos = rindex ($my_string, $my_pattern); # from right

  $q = rindex (“aeiouaeiou”, “o”); # $q= 8

- **Extracting and replacing substrings**

  $s = substr ($my_string, $start, $end);

  $grab = substr (“hola mundo”, 5, 2); # $grab has “mu”

  $grab = substr (“hola mundo”, -3, 2); # $grab has “nd”

- **Advanced classification (sorting)**

```
sub by_number {
        if ($a < $b) {return –1;}
        elsif ($a == $b) {return 0;}
        elsif ($a > $b) {return 1;}
}
@list = (1,2,4,10,20, 32, 126);
@bad_sorted = sort @list; # will have:(1,10,126,2,20,32,4)
@well_sorted = sort by_number @list; # correct order
```

- **Spaceship operator:** < = > instead of **by_number**:

```
sub by_number{
        $a < = > $b;
}
```

or: **@well_sorted = sort {$a < = > $b} @list;** # right order

- Transliteration

$_ = “hola mundo”;

tr/aeoiu/AEIOU/; # $_ has “hOlA mUndO”

tr/a-z/A-Z/; # now $_ has “HOLA MUNDO”

tr/a-z/k/; # now $_ has “kkkk kkkkk”

$_ = “hola mundo”;

tr/a-z/AEIOU/d; # now $_ has “OA UO”

$_ = “hola mundo”;

$num1 = tr/a-z//; # $_ still the same, $num1 has 10

$num2 = tr/a-z/A-Z/; # $_ in uppercase, $num2 has 10

ACCESS TO SYSTEM DATABASES

- File **/etc/passwd**

  getpwnam ($loginname)

  getpwuid ($uid)

Both return:

($log, $pass, $uid, $gid, $quota, $coment, $gcos, $dir, $shell)

- File **/etc/group**

Use: getgrent or getgrid or getgrnam

($log, $pass, $gid, $members)

- **Network** information

  ($name, $aliases, $addrtype, $lenght, @addrs) = gethostbyname($name);

  $name: canonic name

  $aliases: aliases separated by semicolons

  $addrtype: IP type

  $lenght: number of IP addresses

  @addrs: IP addresses

DB manipulation

- First type: DBM
  - Standard **UNIX** library (sendmail, NIS maps)
  - Perl binds a Hash to one DBM
- Opening and closing DBM Hashes

  dbmopen(%ARR, "dbmfile", $mode); # binding a hash

  #dbmfile.dir and dbmfile.pag constitute the DBM

  #$mode: mask of the new file (if it doesn't exist) e.g.: 0644,undef

  dbmclose(%ARR);

- Using the DBM hash

```
$ARR{“one”}= “eins”; #create/update an element
delete $ARR{“three”};
while (($key, $valor) = each(%ARR)){
    print “$key has the value: $ARR{$key}\n”;
}
```

- Second type: DBI/DBD (Database Interface / Database Driver)

  DBI: interface for PERL scripts, independent of the DB

  DBD: driver, specific to the DB, implements the methods from DBI e.g.: **DBD-Oracle**

- **Example**

```
use DBI;

if ( $#ARGV < 0 ) {
        die “Use: select.pl <Database String> <Database Vendor>\n";
}

# Create a new database handle. If it is not possible to get connected: die()
$dbh = DBI->connect( '', $ARGV[0], '', $ARGV[1] );
if ( !defined $dbh ) {
        die “Not possible to get connected to the DB: $DBI::errstr\n"; }
# Prepare a sentence for its execution
$sth =  $dbh->prepare( "
                SELECT id, name
                FROM table" );
```

```
if ( !defined $sth ) {
        die “Not possible to prepare a sentence: $DBI::errstr\n";
}

# Execute the sentence at level of the DB
$sth->execute;

# fetch the rows using a SELECT sentence
while ( @row = $sth->fetchrow() ) {
        print “Returned row: @row\n";
}

# Re-execute the sentence for fetching the rows once again
$sth->execute;
```

```
# Now, store the data in separate variables
while ( ( $id, $name ) = $sth->fetchrow() ) {
        print "ID: $id\tName: $name\n";
}

# Release resources
$sth->finish;

# Disconnect from the DB
$dbh->disconnect;
 exit;
```