Perl--Practical Extraction Report Language

- Developed by Larry Wall – Who is also well-known for the "patch" utility
  - Original purpose: extract data from one source and translate to another format
  - First release in 1987
- An interpreted language (scripting), no need to be compiled
- Support multiple platforms
  - Unix, Linux, MS Window, MacOS, etc.
- It is OPEN SOURCE
- Books
  - Beginning Perl by Simon Cozens, Also available on line at https://www.perl.org/books/beginning-perl/
  - Online doc: https://perldoc.perl.org/

Perl Programming Language

- It pulls together the best features of several other scripting languages
  - **`sed, awk, shells`**, etc.
- Syntax was made up out of
  - C/C++, Pascal, basic, UNIX shells, and others
- A very powerful scripting language
  - System management and text handling with regular expression
  - Database handling
  - CGI programming, dynamic web pages, etc.

First Perl Program: Hello World

```
$ cat hello.pl
#!/usr/bin/perl -w
#This is my first perl program
print "Hello World!\n";
```

- Create a text file `hello.pl` using `vi`
- First line
  - Like shell script, the first line starts with **`#!`** followed by the name of the perl program /usr/bin/perl –w
  - The –w option turns on the warnings, such as undefined variables
- Comment lines start with #
- Lines must end with ";"
- Double quotes are needed to interpret new line, `"\n"`
  - perl does NOT add new line automatically for you as "echo" or print in awk

Program Execution

- First method:
  - Add execution permission to the script

    ```
    chmod  +x script.pl
    ```
  - `./script.pl`
- The other (portable) way
  - using perl program directly: `perl –w script.pl`
- The perl command options
  - `–w`: turn on all the useful warnings
  - `–c`: syntax checking only
- Common syntax errors:
  - Missing semicolon at the end of line
  - Missing prefix: “$”, “@”,or % when define and reference variables

    ```
    $a="Hello World";
    @a=(Hello World);
    %a=("John Smith", 200, "Alex",300)
    ```

Variable Scope

Global Variables vs. Local Variables

- Global Variables
  - They can be “seen” and changed anywhere in the perl script
- Local Variables
  - Declare local variables with keyword “**my**”:
  - They are constrained to the enclosing block and all the blocks inside/beneath it

```
my ($lname, $fname);
my $lname='Lin';
my $fname='Amy';
```

Perl Pragma: “strict”

- Generally, you can use variables on the fly, but very error-prone
- **`strict:`** the most important perl pragma
  - The “strict” progama forces the declaration of variables via the use of the Perl keyword **`my`**
  - Add line **`use strict`** at the beginning of the script (after the first line and commenting lines)
    - “use” is the key word for import a perl modules (similar to Java’s **`import`**)
- In case you need to define a global variable within a subroutine (function), use key word “our” instead of “my”

Quoting

- Single quotes, double quotes, and back slash quotes
- Quoting rules are the same as in Shell scripting
  - No interpretation for everything inside single quotes
  - You will need use single quotes or \ if double quotes are needed
- Examples
  - `print “$a\n”;`
  - `print ‘$a\n’;`
  - `print $a, “\n”;`
  - `Print “3+4=”, 3+4, “\n”;`
- Double quotes are used more often than single quotes
  - How to print the following line to the screen using Perl

    He said: "I am 90 years old!"

```
1.print 'He said: "I am ', $year, ' old!"', "\n";
2.print "He said: \"I am $year old!\" \n";
```

Data Type #1: Scalar

- A scalar can be a number or a string, **prefix "$"**
  - **$name="Jone Smith";**
  - **$number=10;**
- Operations
  - String Concatenation using a "dot"
    - **$a = "hello"."world";**
    - **$a = $b."Hello"."World";**
  - Arithmetic Operations
    - **+ - * / **** (power), bitwise operations
      - **$num = (4**2 + $salary/5);**
    - Numbers are all treated as floating numbers
- String and number conversion, Perl will do this for you
  - **$num="0.25"; $value=4.0 * $sum**

Data Type #2: Regular (Numeric) Array (List)

- An array variable is assigned with prefix @
  - Index starts at “0”
- Define an array
  - elements separated by comma, enclosed with parentheses
    - `@a = (1, 2, 3); @b = (“hello”, “world”, “!”);`
    - `@c = (@a, @b, “Good Day”);`
  - **`Using qw subroutine (function)`**
    - `@b = qw(hello world !);` (“qw” refers to quote word)
- **`split`**: split a string into a list:
  - **`split(pattern, string, limit)`**
    - `@chars = split(//, $word);`
    - `@words = split( /, $sentence);`
    - `@sentences = split(/\./, $paragraph);`
    - `@paragraphs= split(/\\n/, $essay);`

- Reference element of an index array
  - `$a1 = $a[0];`

  (element is a scalar, needs “$”)
- Length of an array: **`$#varname + 1`**
  - **`$#varname`** : the index of the last element
- Array to string with “join”
  - `@names=(“John”, ‘Michael’, “Jessie Smith”);`
  - `$names=join(‘;’, @names);`
- Print array
  - `print “@names\n”;`

Command-line Arguments

- **@ARGV** ➔ A predefined regular array contains the command-line arguments
- **$#ARGV+1** ➔ The total number of arguments ($#ARGV gives the index of the last element, index starts with 0)

```
#!/usr/bin/perl -w
use strict;
#use hello_to.pl;
if($#ARGV+1 == 0) # equivalent to if($#ARGV==-1)
{
    print "Usage: hello arg1 arg2 ...\n";
    exit 1;
}
for( @ARGV ){ print $_,"\n";}
for(my $i=0; $i<= $#ARGV; $i++){ print $ARGV[$i],"\n";}
my $name;
foreach $name (@ARGV){ print $name,"\n";}
```

Some String Functions

chop(string)	Removes the last character of the string
chomp (string)	remove the last end of line character, otherwise, do nothing
chr(number)	Returns the character having the ASCII number
join(string, array)	Returns a string that consists of all the elements of array joined with string
split(pattern, string, limit)	Split the string to array up to limit with pattern as the delimiter,
lc(string)	Return string in lower case
uc(string)	Return string in upper case
length(string)	Returns the length of the string
substr(string, offset, len)	Returns portion of the string starting at offset, up to length len
reverse (@list)	Reverse the order of the elements in the list

Regular Array Operations

- shift
  - Remove a value from the front of the array
  - `$first = shift(@_);`
  - `$first = shift(@_);`
- unshift
  - Add an element at the front of the array

    `unshift($namelist, "Smith");`
- push: append element to the end of an array
  - @namelist = ("John", "Jones");
  - push(@namelist, 'Smith');
- pop: remove element from the end of an array
  - $name = pop(@namelist);

Sorting

- Sorting alphabetically
  - Ascend order (default)

```
@sortedList = sort(@list);
```

  - Descend order (reversed)

```
@sorted = sort{$b cmp $a} (@list);
```

- Sorting numerically for elements of numbers
  - Ascend order (default)

```
@sortedList=sort{$a<=>$b} @list;
```

  - Descend (reversed)

```
@sorted_list = sort { $b <=> $a } @list;
```

Array to String

Convert regular array to a string with "join" function

For array @fruit

```
@fruit=(apple, pear, peach);
```

To:

- `$strfruit = join(" ", @fruit);`
- `$strfruit = join("\n", @fruit);`

Data Type #3: Associative Arrays (Hashes)

- Variable prefixed with `%`
- Creation: pairs of **`key` ➔`value`**
  - **`%salary = (“David”,10000, “John”,2000);`**
  - **`%grade = ( "David" => 100,`**
    `"John" => 90`
    `);`
- Reference an element with curly bracketed key
  - **`$salary_david=$salary{“David”};`**

Note:

- How to reference an element of a regular (index) array?
- How to reference element of an associate array in awk?

Some Special Default Variables

- **`@ARGV`**
  - Command line arguments when running the perl script
- **`%ENV:`** Predefined hash for environment variables.
  - **`$path = $ENV{'PATH'};`**
  - **`$username=$ENV{"LOGNAME"};`**
- **`$_`**
  - Default variable for function arguments and pattern searching space
    - In many functions, if an argument is not specified, $_ will be automatically assigned
    - **`chop ($_)`** equivalent to **`chop`**
- **`@_`**
  - Array: the arguments passed to a function.
  - It is a local variable to that function

Working with Associative Arrays

- Add new element to an exiting hash array
  - `$salary{"Eric"} = 3000;`

  Note, if key Eric exists, the previous value will be replaced by the new one (modify an existing element)
  - `$salary{"Eric"} = 2000;`
- Delete an existing entry/key
  - `delete $salary{"John"}`
- Reset an associative array
  - `%name=() or undef %name;`
  - `undef $name; undef @name;`

Functions for Hash Array

- **exists**
  - To check if a key exists

    if (not exists $name{“key”})….
- **keys**
  - Get the list of keys of the associative array

```
@keylist = keys %salary;
```

- **values**
  - Get the list of values of the hash array

```
@valuelist = values %salary;
```

❖ each

Returns a two-element list that contains a key and value pair from the given associative array one by one

- **($name, $value) = each %salary**

```
while (($name,$value)=each %salary)
{   print "$name = $value\n";}
```

Note: "each" returns false when the end of array is reached

for & foreach loop

- Regular iterator
  - `for ($i=0; $i<=10; $i++) { … }`
- Regular arrays
  - `for (@names) { print "name is $_\n";}`
  - Special variable `$_` is often used as the 'default variable', whatever it is passed to
- foreach loop
  - `foreach $name (@names)`
    `{ print "It is $name\n";}`
  - `foreach $value (values %salary)`
    `{ print " $value\n");}`
  - `foreach $name (keys %salary)`
    `{ print "$name: $salary{$name}\n");}`

do/while/do/until

```
while ($num < $total) {
    $num++;
}
```

```
until ($x > 6) {
   print "num is $x!\n";
   $x++;
}
```

```
do {
    $junk++;
 } while ($junk < 39);
```

```
do {
    $junk++;
 } until ($junk >39);
```

- Two keywords for loop control
  - **next**: Skip to the next iterator
  - **last**: To exit the loop

Logical Operators

- Compare Numbers
  - `$a > $b`
  - `$a < $b`
  - `$a == $b`
  - `$a != $b`
- Compare Strings
  - `$a gt $b` ➔$a sorts alphabetically after $b
  - `$a le $b` ➔ $a sorts alphabetically before $b
  - `$a eq $b` ➔ $a is the same as $b
  - `$a ne $b` ➔ $a is not the same as $b

Logical Conjunctions

- **`$a and $b  ; $a && $b`**
  - True if both $a and $b are true
- **`$a || $b ; $a or $b`**
  - True if either $a or $b is true
- **`not $a`**
  - True if `$a` is not true

if/elsif/elsif/else

```
if ($a < $b) { $z++; }
elsif ($b < $c) { $y++; }
elsif ($c < $d) { $x++; }
else {$yikes++;}
```

***Note: you have to have the curly braces even it contains only one line

Regular Expressions

- Matching
  - `if ($name =~ /John/) { …}`
  - If hinting special variable **`$_`**
    - `if (/John/) {….}` or
    - `if (/John/i) {…}` # if you want to ignore case
  - Matching a variable's value, the variable will be interpolated to its value
    - `if (/$name/)`
- Modifying strings with regex substitution
  - `s/John/John has retired/; print $_, "\n";`
  - `$name =~ s/John/Alex/; print "$name\n"`

split & join

- “split” will split a string with /delimiter/ into a list (array)
  - `split ⇔ split(/  */, $_) ⇔ @_=split (/\s+/, $_)`
- “join” will convert a list into a string, put a defined delimiter between the elements
  - `join “_”, @wordlist; print “$_\n”; ⇔`
  - `$sentence = join “ ”, $wordlist;`

Meta Characters and More

Learn more about regular expressions and meta characters from:

http://blob.perl.org/books/beginning-perl/3145_Chap05.pdf

Standard Input/Output

- STDOUT

  `print STDOUT "name:";`←→`print "name:";`

  - `STDOUT` is normally no need to be specified
- STDIN, the stream name is enclosed with `<>`
  - STDIN can be dropped too

    `$passwd = <STDIN>;` ←→ `$passwd = <>;`
  - Reading through a loop
    - `while (<STDIN>) { print "$_\n";}`
    - `while (<>) { print "$_\n";}`
    - `$_`: what just being read
  - `chomp` is normally followed after STDIN to remove the last EOL character
    - `chomp $passwd;` # remove the EOL
    - chomp # will be the default $_

File Input/Output

- I/O with files with open and file redirections: **<, >, >>**

```
open( IN, "</some/directory/file.in");
open ( OUT, ">/some/directory/file.out");
# in case to append to the outfile, use >> instead
while (<IN>) { print OUT "$_\n"; }
close(IN);
close(OUT);
```

- When open file for read, "**<**" dropped for simplicity
  - `open(IN, "file.in"); # open file for read`
- Upper-case letters for file descriptors by convention

I/O Error Handling

- The "`die`" function
  - Program exits in case of error

```
open (IN, "/some/directory/file.in")
  || die "cannot open: $!";
```

  - `$!` Contains the most recent system error
  - Program exits if open file fails

Example

```
#!/usr/bin/perl -w
$thisd=$ENV{“PWD”};
open(AC, "< $thisd/info.txt") || die “Error $!\n";

while (<AC>)
{
    chomp;
    if ((/^#/)||(/^$/)||(!/=/)) { next; }
    $_ =~ s/\"//g;
    if (/^YEAR)/)
    {
       ($foo, $year) = split(/=/);
       last;
    }
}
close(AC);
```

Perl Subroutines

- Function is defined with “sub”
  - `sub print_name { my $name=“Smith”; …}`
- Global variables are “visible” to subroutines
  - In case of “use strict”, global variables have to be declared with keyword, `our,` otherwise, it is not available outside the subroutines
- Invoke a subroutine

```
print_name($lastname, $firstname);
```

- Parameters passed to a function: @_
- @_ is a “list”
  - The list is saved in the special default variable: @_
  - sub print_name { ($last, $first) = @_; print “$last, $first\n”; }
    - You can have more elements than @_ has, the rest just empty string
    - Ex: ($last, $first, $middle) = @_; $middle will be no value

Perl Subroutines (Cont'd)

- Subroutines can be declared first, defined later. They have to be declared or defined before being referenced.
- Subroutines can return a value explicitly or implicitly
  - Explicitly return a value using return statement
  - Implicitly, the last "thing" of the subroutine is returned
  - Function can return
    - a scalar,
    - a list, or
    - an associative array
- Subroutines can use pass by reference to pass and return values…

A perl script to compute the max number of a list

```
sub max
{ my $maxValue = shift @_;
  foreach (@_)
  {
      if ($_ > $maxValue)
      { $maxValue = $_; }
  }
  return $maxValue;
}
```

`max` takes a list of numbers

```
print max(@array);
```

Directories

- Change directory:
  - `chdir("/some/path") || die "Cannot chdir to /some/path ($!)";`
- Directory handles
  - Access content of directory with `readdir`

```
opendir(DIRHANDLE, "/some/path") || die
  "Cannot opendir /some/path: $!";
foreach $name (sort readdir(DIRHANDLE))
  { print "found file: $name\n"; }
closedir(DIRHANDLE);
```

Manipulating Directories

- Create and remove directories
  - mkdir("newdir", 0755) || die "Cannot mkdir newdir: $!";
  - rmdir("olddir") || die "Cannot rmdir olddir: $!";
- Rename file/dir
  - `rename("file.txt", "file-old.txt") || die "Cannot rename file.txt: $!";`
- Remove file(s)
  - `unlink($filename)`
  - `unlink (@filelist);`
- Testing

```
if (-d "/some/path") { $where = "/some/path"; }
else { $where = "/another/path"; }
```