Lecture 5/6: Scripting and Perl

COMP 524 Programming Language Concepts
Stephen Olivier
January 29, 2009 and February 3, 2009

Based on notes by N. Fisher, F. Hernandez-Campos, and D. Stotts
Goal of Lecture

• Discuss background on Scripting languages and Perl.
Origin of scripting languages

• Scripting languages originated as job control languages
  • 1960s: IBM System 360 had the “Job Control Language”
  • Scripts used to control other programs
    • Launch compilation, execution
    • Check Return Codes

• Scripting languages became increasingly powerful in UNIX
  • Shell programming, AWK, Tcl/Tk, Perl
  • Scripts used to “glue” applications
System Programming Languages

- System languages (e.g., Pascal, C++, Java) replaced assembly languages.
  - Two main advantages:
    - Hide unnecessary details (high level of abstraction)
    - Strongly Typed.
Strongly vs Weakly Typed Languages

- Under Assembly, any register can take any type of value (e.g., integer, string).

- Under **Strongly Typed languages**, a variable can only take values of a particular type.
  - For example, “int a” can only have values of type “integer”
Strongly vs Weakly Typed Languages

- Weakly Typed languages infer meaning at run-time
  - **Advantage**: Increase Speed of development.
  - **Disadvantage**: Less error checking at compile time.
- Not appropriate for low-level programming or large programs
Typing and “Degree of Abstraction”

- **Assembly**
- **C**
- **C++**
- **Java**
- **Tcl/Perl**
- **Visual Basic**
- **Scripting**

**Instructions/Statement (Level of Abstraction)**

- None
- Strong

**Degree of Typing**
Perl (Practical Extraction and Report Language)

• Larry Wall Created Perl in late 80s
  • Originally designed to be more powerful than Unix scripting.
  • Wanted “naturalness” ... shortcuts, choices, defaults, flexibility.

• Perl is dense and Rich
  • “Swiss-army chainsaw”
  • “Duct tape for Web”
  • “There is more than one way to do it!”
  • Often experienced Perl programmers will need a manual when reading other people’s code.
What Perl Does Well

• String Manipulation
• Text Processing
• File Handling
• Regular Expressions and pattern matching
• Flexible arrays and hashes
• System Interactions (directories, files, processes)
• CGI scripts for Web sites
Perl Overview

• Perl is **interpreted**.

• Every statement ends in a **semicolon**

• Comments **begin with “#” and extend one line**
  
  • We’ll see how to do multi-line comments later

• What Perl doesn’t do well:
  
  • Complex algorithms and data structures.
  
  • Well defined and slowly changing functions.
Built-in Data types

- No type Declarations
- Perl has three types:
  - Scalar
  - Array
  - Hash (Associative Array)
- Integers, float, boolean, etc... are all of type Scalar.
Built-in Data Types: Scalar

- Scalars begin with “$”
- Can take on any integer, real, boolean, and string value

```php
$A = 1;
$B = "Hello";
$C = 3.14;
$D = true;
```

- There is a default variable “$_”
Scalars in Strings

• To use a scalar in a string simple insert it!

```bash
$A = 1;
print ("A’s value is $A \n");
```
Addition and Concatenation

• To **add** two scalars together, we use “+”

```
$A = 1;
$B = 2;
$C = $A + $B;
```

• To **concatenate** two strings together, we use “.”

```
$A = “hi”; 
$B = “bye”; 
$C = $A . $B;
```
Context

- When a scalar is used, the value is converted to the appropriate context:

```
$A = "hi";
$B = 3;
$C = $A . $B; #C = "hi3"
```

```
$A = "hi";
$B = 3;
$C = $A + $B; #C = "3"
```

```
$A = "4";
$B = 3;
$C = $A . $B; #C = "43"
```

```
$A = "4";
$B = 3;
$C = $A + $B; #C = "7"
```
Built in Data type: Array

• Array variables begin with “@”

```
@A;
```

• Using “=(xxx,yyy,zzz,...)” we can define the content of the array

```
@A = (1, “two”, 3.13, true);
```

• Using $foo[xxx] we can access individual elements of the array @foo.

```
print ($A[1]); #Prints “two”
```
Built in Data type: Array

• Using "#$foo" we can get the max index of the array "@foo"

```perl
@A = (1, "two", 3.13, true);
print $#A; #Prints 4
```

• There is a default array "@_"
Built in Data Types: Hash

- Hashes are like arrays, except that they are indexed by any scalar type, not just integer.
- Hash variables begin with “%”
- Can be defined as via “( ‘index-1’, value-1, ‘index-2’, value-2,...)

  %A = (‘first’, 1, ‘junk’, ‘value’, 3.14, true);

- Subscripts are accessed by “{}” and can be any scalar

  print $A(3.14); #Prints “true”
Built in Data Types: Hash

- Great for text processing
  - Building tables, lists, etc....
- Built-in function “keys” gets all subscripts.

```perl
%A = ('first', 1, 'junk', 'value', 3.14, true);
foreach (keys (%A)) { #Loads values in t "$_"
    print "($A{$_}):$_ \n";
}
```
Control Flow

```php
$b = 3;
if($b < 10){
    $a = 5;
} elseif ($b < 20){
    $a = 15;
} else {
    $a = -3;
}
print($a);
```

```php
$c = 3;
print($c >= 10 ? 20 : 10). "\n";
```
Control Flow

```php
while($d<37){
    $d++;
    $sum += $d;
}
```

```php
until($d>=37){
    $d++;
    $sum += $d;
}
```
Control Flow

do{
    $d++;
    $sum += $d;
} while ($d<37);

do{
    $d++;
    $sum += $d;
} until ($d>=37);

Foreach

@group = ("red", "blue", "green", "tan");
foreach $item(@group){
    print "$item \n";
}

Files and I/O

open(INDATA, "index.html"); #reading

open(INDATA, ">index.html"); #writing

open(INDATA, ">>index.html"); #appending

open(INDATA, "index.html") || die "Error";
close(INDATA);
open(INDATA, "index.html");
$in = <INDATA>; #Gets one line as a scalar
@all_in = <INDATA>; #Gets all lines as an array
#all_in[0] = first line
#all_in[1] = second line
close(INDATA)
Files and I/O

``` perl
open(INDATA, "index.html")
foreach $line(<INDATA>) {
    print $n++.": $line";
}
close(INDATA);
```
Files and I/O

```plaintext
open(OUTDATA, ">index.html")
print OUTDATA "Out";
close(OUTDATA);

print STDOUT "Out";
```
Subroutines

```perl
sub aFunc{
    my($a, $b, $c); #makes $a, $b, and $c local
    $a = $_[0];    #Set’s a to first input
    $b = $_[1];    #Set’s b to second input
    $c = $a + $b;
    print $c . "\n";
    return "done\n";
}
```
Subroutines

print &aFunct(12,5);
$retValue = &aFunc(12,5);
aFunc(12,5);
$x = noArgs();
$x = &noArgs;
Regular Expressions

/.at/ #matches “cat”, “bat”, but not “at”
/[aeiou]/ #matches single character
/[0-9]/ #match one char
/[0-9]*/ #match zero or more chars from range
/[^[0-9]/ #match zero or more chars NOT in range
/c*mp/ #“cccmp”, “cmp”, “mp”, NOT “cp”
/a+t/ #“aaat”, “at”, “t”
/a?t/ #zero or one “a”s, “at” or “t” not “aaaat”
/^on/ #start... “on the” NOT “the on”
on$/ #end... “the on” not “on the”
cat/i #ignore case
/\*\*\*/ #match “**”
Regular Expressions

• By default, applied to "$_" scalar

```perl
$_ = "Hello World";
if (/Hello/) { print ("Hello in \$_\n"); }
```

• Can be applied to other scalars via "=~"

```perl
$a = "Hello World";
if ($a =~ /Hello/) { print ("Hello in \$_\n"); }
```
Regular Expressions

• Replace “foo” with “bar” by “s/foo/bar/”

```
$a = "Hello World World";
$a =~ s/World/Mars/;
print ($a . "\n"); #Print “Hello Mars World”
```

• Only works for first match.

• To apply to all use “s/foo/bar/g”

```
$a = "Hello World World";
$a =~ s/World/Mars/g;
print ($a . "\n"); #Print “Hello Mars Mars”
```
Regular Expressions

- Replace regardless of case use “s/foo/bar/i”

```perl
$a = "Hello World World";
$a =~ s/world/Mars/i;
print ($a . "\n"); #Print “Hello Mars World”
```

- Combine with “global”

```perl
$a = "Hello World World";
$a =~ s/world/Mars/gi;
print ($a . "\n"); #Print “Hello Mars Mars”
```
Pattern Matching and Input

while (<>){ #Puts “Standard Input” into $_
  if(/chicken/){
    print "Chicken found :$_";
  }
} #Prints “For each
System Interactions

• Run a system command foo use system(“foo”);

  system(“ls”); #runs “ls”

• To get return from system use “backticks” ( ` )

  $retVal = `pwd`;
  print “$retVal\n”; #Prints working Dir.
Pipes

• Open a pipe as a filehandle

```perl
$pid = open(DATAGEN, "ls -lrt |") || die "oops\n";
while(<DATAGEN>){ print; }
close(DATAGEN) || die "oops again\n";
```

• Pipe from a process

```perl
$pid = open(SINK, "| more") || die "oops\n";
$a = `ls`;
print SINK $a; # Pipes output from "ls" into "more"
close(SINK) || die "oops again\n";
```
Eval

- Perl scripts can invoke another copy of the perl interpreter to evaluate functions during execution (via the eval function)

```perl
$str = '$c = $a + $b';
$a = 10; $b = 15;
eval $str; #Evaluates $str
print "$c\n";
```
Eval

- Eval can be used to make a “mini-Perl” interpreter

```perl
while(defined($exp = <>)){
    $result = eval $exp;
    if($@) { #Check for Error Message
        print "Invalid input string:
 $exp";
    } else {
        print $result. "\n";
    }
}
```
Eval: BE Careful

• If the following program were run...

```perl
$exp = <>;
$result = eval $exp;
```

• ...with the input “system(“cd /; rm -r*”);”

• Then the **hard drive would be erased**!
Examples

• Suppose we want to process a text file with the following methods

  • Any Line containing “IgNore” will not go to output
  • Any line with “#” will have that char and all after it removed.
  • Any string “*DATE*” will be replaced with the current date
  • All deleted lines (and partial lines) will be saved in a separate file.
```perl
$inf = "foo.txt" ; $OUTF = "bar.txt" ; $scpf = "baz.txt" ;
open(INF,"<$inf") || die "Can't open $inf for reading" ;
open(OUTF,">$OUTF") || die "Can't open $OUTF for writing" ;
open(SCRAPS,">$scpf") || die "Can't open $scpf for writing" ;
chop($date = `date`) ; # run system command, remove the newline at
# run system command, remove the newline at
the end
foreach $line (<INF>) {
   if ($line =~ /IgNore/) {
      print SCRAPS $line ;
      next;
   }
   $line =~ s/\*DATE\*/$date/g ;
   if ($line =~ /\#/) {
      @parts = split ("#", $line);
      print OUTF "$parts[0]\n" ;
      print SCRAPS "#" . @parts[1..$#parts] ; # range of elements
   } else {
      print OUTF $line ;
   }
}
close INF ; close OUTF ; close SCRAPS ;
```
Another Example

• Consume an input file and produce an output with duplicate lines removed

```perl
open(INF,"<foo.txt");
foreach (<INF>) {print unless $seen{$_} ++; }
```
Another Example

• Consume an input file and produce an output with duplicate lines removed (and alphabetizes them!)

```perl
open(INF, "<foo.txt");
foreach (<INF>) {
    $unique{$_} +=1;
}
foreach (sort keys(%unique)) {
    print "($unique{$_}):$_";
}
```
Large comments

Large comments can be constructed by using "=comment" and "=cut"

```python
print("a");
=comment
print("b");
=cut
print("c\n"); #Prints "ac"
```
CPAN

- Comprehensive Perl Archive Network (CPAN) contains lots of useful Perl modules.
  - www.cpan.org
References (not bibliography.... “pointers”)

- References are scalars.
- A reference to $foo, “$rfoo$, is defined as “\$foo”.
- The value of $foo is retrieved via “$$rfoo$”.

```php
$a = 3; $b = $a; $ra = \$a;
$a = 4;
print $$ra . " " . $b;
#prints “4 3”
```
References (not bibliography.... “pointers”)

- Arrays and hashes are similar
- Can get with “$$” or “->”

```perl
@arr = (10,20,30);
%hsh = ("fisrt", 10, "sec", 2);
$rarr = @arr;
$rhsh = %hsh;
print($$rarr[0] . " " . $$rhsh{"sec"});
print($rarr->[0] . " " . $rhsh->{"sec"});
# prints “10 2”
```
Arrays of references

- Can make an array of references

```perl
@arr1 = (10,20,30);
@arr2 = (40,50,60);
@rar = (@arr1, @arr2);
print("$rar[0][0] $rar[1][2]\n");
# prints “10 60”
```