CSCI-GA.3033.003
Scripting Languages

6/7/2012

Textual data processing (Perl)
Announcements

• Homework 2 due Friday at 6pm.

• First prelim 9/27, Review on 9/25

• Additional TA:
  – Theodoros Gkountouvas

• Possible room change in the future:
  – Watch Piazza for announcements
Outline

• Perl Basics
About Perl

• Practical Extraction and Reporting Language
  – Regular expressions
  – String interpolation
  – Associative arrays

• TIMTOWDI
  – There is more than one way to do it
  – Make easy jobs easy
  – … without making hard jobs impossible
  – Pathologically Eclectic Rubbish Lister
## Orthogonality

<table>
<thead>
<tr>
<th>Definition of orthogonal</th>
<th>Language design principle</th>
<th>Violation of orthogonality</th>
</tr>
</thead>
<tbody>
<tr>
<td>At right angles (unrelated)</td>
<td>Uniform rules for feature interaction</td>
<td>VBA object assignments</td>
</tr>
<tr>
<td>Not redundant</td>
<td>Few, but general, features</td>
<td>VBA positional vs. named args</td>
</tr>
</tbody>
</table>

Perl is **diagonal** rather than **orthogonal**:

“If I walk from one corner of the park to another, I don’t walk due east and then due north. I go northeast.” [Larry Wall]

⇒ shortcut features even when not orthogonal
Related Languages

• Predecessors: C, sed, awk, sh
• Successors:
  – PHP (“PHP 1” = collection of Perl scripts)
  – Python, JavaScript (different languages, inspired by Perl’s strengths + weaknesses)
• Perl 5 (current version, since 1994)
• Perl 6
  – Larry Wall has been talking about it since 2001
  – Evolved into a separate language
How to Write + Run Code

- `perl [-w] -e 'perl code'`
  - “-w” flag produces warnings
- `perl [-w] script.pl`
- `script.pl`
  - Write the file in Vi or Emacs or ...
  - `chmod u+x script.pl`
    - Makes script executable
  - `#!/usr/bin/perl -w`
    - In first line of script specifies interpreter
- `perl [-w] -d -e 42`
  - Edit-eval-print loop (debug the script “42”)

Perl
Lexical Peculiarities

• Single-line comments: #
• Semicolon required after statements unless {last; in; block}
• Quotes around certain strings (bare words) optional in certain cases (e.g., as hash key)
• v-string: v13.10 = "\x{13}\x{10}"
• Interpolation; pick-your-own-quotes; Heredocs; POD (plain old documentation)
• Many more…
Types

Perl

- scalar
  - numeric
    - int
    - double
  - string
  - void
  - function reference
  - reference
    - UNIVERSAL
      - (blessed references)
  - file handle
  - (composite)
  - array
  - hash
    - (other references)
Sigils, a.k.a. “Funny Characters”

- Symbol that must appear in front of variable, showing its type
  - $=scalar, @=array, %=hash, &=function, *=typeglob
  - E.g., $a[0] is element 0 of array @a
- Unlike shell, Perl requires sigil also on left-hand side of assignment
- @{$id} is the same as $id
- Function sigil & not required for call
## Perl Variable Declarations

<table>
<thead>
<tr>
<th>Scope Type</th>
<th>Example Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit</td>
<td><code>print $a + 1; $b = 5;</code></td>
<td>Read <code>undef</code> if non-existent</td>
</tr>
<tr>
<td>Local, lexical scope</td>
<td>my $c; my ($d,$e)=(3,4);</td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td>sub f{ our $g; print $g++; }</td>
<td>Hides locals; unlimited lifetime</td>
</tr>
<tr>
<td>Local, dynamic scope</td>
<td>sub h{print $i;} sub k{ local $i=5; h }</td>
<td>Can also localize single array/hash item</td>
</tr>
</tbody>
</table>
# Concepts

## Static vs. Dynamic Scoping

<table>
<thead>
<tr>
<th>Static scoping</th>
<th>Dynamic scoping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound in closest nesting scope in program text</td>
<td>Bound in closest calling function at runtime</td>
</tr>
</tbody>
</table>

```perl
#!/usr/bin/perl -w
$x = 's';
sub g {
    my $x = 'd';
    return h()
}
sub h {
    return $x
}
print g(), "\n"; #s
print $x, "\n"; #s
```

```perl
#!/usr/bin/perl -w
$x = 's';
sub g {
    local $x = 'd';
    return h()
}
sub h {
    return $x
}
print g(), "\n"; #d
print $x, "\n"; #s
```
Interpolation

• Expansion of values embedded in string
• Single-quoted string literal 'abcde'
  – Only interpolate \ and \n
• Double-quoted string literal "abcde"
  – More escape sequences, e.g., \n
  – Variables only, starting with @ or $
  – Use curleys to delimit: "time \{hours\}h"

• Trick to interpolate arbitrary expressions
  – "... @{[expr]} ..." or "... @{[[scalar expr]]} ..."
<table>
<thead>
<tr>
<th>Operator</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code>, &quot;...&quot;, <code>\</code>..., print, sort, ...</td>
<td>L</td>
<td>Terms, function call, quoting, list operators (leftward)</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>2</td>
<td>Dereference and member access</td>
</tr>
<tr>
<td><code>++</code>, <code>--</code></td>
<td>1</td>
<td>Auto-increment, auto-decrement</td>
</tr>
<tr>
<td><code>**</code></td>
<td>2</td>
<td>Exponentiation</td>
</tr>
<tr>
<td><code>!, ~, \ , +, -</code></td>
<td>1</td>
<td>Negation (!, ~, -), reference (), no-op (+)</td>
</tr>
<tr>
<td><code>=~</code>, <code>!~</code></td>
<td>2</td>
<td>Binding to regular expression pattern match</td>
</tr>
<tr>
<td><code>*, /, %, x</code></td>
<td>2</td>
<td>Multiplicative (x is string repetition)</td>
</tr>
<tr>
<td><code>+</code>, <code>-</code>, <code>.</code></td>
<td>2</td>
<td>Additive (dot is string concatenation)</td>
</tr>
<tr>
<td><code>&lt;&lt;</code>, <code>&gt;&gt;</code></td>
<td>2</td>
<td>Bitwise shift</td>
</tr>
<tr>
<td><code>eval, sqrt, -f, -e, ...</code></td>
<td>1</td>
<td>Named unary operators, file test operators</td>
</tr>
<tr>
<td><code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>, <code>lt</code>, <code>gt</code>, <code>le</code>, <code>ge</code></td>
<td>2</td>
<td>Relational (<code>lt</code>, <code>gt</code>, <code>le</code>, <code>ge</code> is for strings)</td>
</tr>
<tr>
<td><code>==</code>, <code>!=</code>, <code>&lt;=&gt;</code>, <code>eq</code>, <code>ne</code>, <code>cmp</code></td>
<td>2</td>
<td>Equality (<code>eq</code>, <code>ne</code>, <code>cmp</code> is for strings) (<code>&lt;=&gt;</code>, <code>cmp</code> yield -1/0/1)</td>
</tr>
<tr>
<td><code>&amp;</code>, `</td>
<td><code>, </code>^`</td>
<td>2</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>2</td>
<td>Logical and (short-circuit)</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td><code>, </code>//`</td>
</tr>
<tr>
<td><code>...</code>, <code>...</code></td>
<td>2</td>
<td>Range (in list context) or bistable (in scalar context)</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>3</td>
<td>Ternary conditional</td>
</tr>
<tr>
<td><code>=</code>, <code>+=</code>, <code>-=</code>, <code>*=</code></td>
<td>2</td>
<td>Assignment; return l-value of target</td>
</tr>
<tr>
<td><code>,</code>, <code>=&gt;</code></td>
<td>2</td>
<td>List (in list context) or sequencing (in scalar context)</td>
</tr>
<tr>
<td><code>print, sort, ...</code></td>
<td>N</td>
<td>List operators (rightward)</td>
</tr>
<tr>
<td><code>not, and, or, xor</code></td>
<td>2</td>
<td>Logical (short-circuit; not all same precedence)</td>
</tr>
</tbody>
</table>
Operators: List vs. Named Unary

• Different precedence rules
• List operator (most user-defined functions)
  – High leftward, low rightward precedence
  – \(@a = (1,5,\text{sort} 9,2); \text{print @a}; \#1529\)
• Named unary operator
  – Lower than arithmetic, higher than comparison
  – \(@a = (1,5,\sqrt{9},2); \text{print @a}; \#1532\)
• Call either one with parentheses
  – Highest precedence
  – \(@a = (1,5,\text{sort}(3+6),2); \text{print @a}; \#1592\)
**Perl**

### Input and Output

- **Output**
  - `print "Hello, world!";`
  - `print STDERR "boo!";`
  - `printf "sqrt(%.2f)=%.2f\n", 2, sqrt(2);`

- **Input**
  - `$lineFromStdIn = <>;`
  - `open MYFILE, '<recipe' or die "$!";`
  - `$lineFromMyFile = <MYFILE>;`
  - `@allLines = <MYFILE>;`
Arrays

• Resizable
• Literals: list \(@a=(1,3,5)\), range \(@b=2..4\)
• Indexing: e.g. \($a[1]\)
  – Zero-based; negative index counts from end
  – $\#a$ returns last index of @a, in this case, 2
  – Write to non-existent index auto-vivifies
• Free: \(\texttt{undef}\) @a, truncate: $\#a=1$
• Array slice: using multiple indices, e.g., @a[0,2] or @a[1..2]
• Using array in scalar context: returns length
  – \(\texttt{scalar @}a\); \# 3 = size
Perl Poetry

```perl
#!/usr/bin/perl -w
while ($leaves > 1) {
    $root = 1;
}
foreach($lyingdays{‘myyouth’}) {
    sway($leaves, $flowers);
}
while ($i > $truth) {
    $i--;
}
sub sway {
    my ($leaves, $flowers) = @_;
    die unless $^O =~ /sun/i;
}

Though leaves are many, the root is one;
Through all the lying days of my youth
I swayed my leaves and flowers in the sun;
Now I may wither into the truth

Wayne Myers port of the Yeats poem,
“*The Coming Of Wisdom with Time*”
```
• Today’s lecture
  – Basics of Perl

• Nest lecture
  – Associative arrays
  – Regular expressions