WeBWorK PREP Webconference

Paul Pearson
Fort Lewis College
May 26, 2011

A. Preliminaries about Perl

1. Webwork is built from Perl

- advantages: scripted language, popular, fast, etc.
- disadvantages: sometimes tricky syntax (unavoidable?), restrictive data types
- specialization: Perl → PG (Problem Generation) → MathObjects

2. Purpose of Webwork

- Deliver questions to students in two display modes:
- HTML output
- PDF output

3. Data types in Perl

- # is the comment character
- ; ends a line of code
- Perl has scalars, which are strings or numbers. Named scalars start with $.

```
$name = “Paul Pearson”;
$num = -5;
```

3. Data types in Perl

- Perl has arrays of scalars. Named arrays start with @.

```
@birds = (“robins”,”blue jays”,”cardinals”);
@numbers = (-4, 3.14, 1000);
```

3. Data types in Perl

- To access a scalar inside an array, use
  ```
  $birds[0];
  $numbers[1];
  ```
  Notice that we used $, not @, when accessing a scalar inside an array. Also, the first entry of any array has index 0, not 1, so $birds[0] has the scalar value robins, while $numbers[1] has the scalar value 3.14.

3. Data types in Perl

- You can get the index of the last element in an array using one of these:
  ```
  $#birds;
  scalar(@birds);
  ```
  both of which will return 2. Notice that the number of elements in this array is 1 more than the index of the last element.

3. Data types in Perl

- You can slice an array to create another array:
  `@basballteams = @birds[1..2];`
  will create an array @baseballteams with elements “blue jays” and “cardinals”.

3. Data types in Perl

- Perl also has hashes (associative arrays of scalars), which we won’t talk about right now.

4. Arithmetic in Perl

- Operations: +, -, *, /, ** (exponentiation), % (modular arithmetic / remainder)
- Gotcha 1: Juxtaposition does not mean multiply:

```
5 * 2;   # correct
(5)(2); # incorrect
5 2;     # incorrect
```

4. Arithmetic in Perl

- Gotcha 2: ^ is the shift operator, not exponentiation

```
5**2; # correct exponentiation
5^2;  # incorrect
```

4. Arithmetic in Perl

- Gotcha 3: -- (minus minus) is the decrement operator, e.g., 5-- is the same as 4. Correct way: use extra space or parentheses:

```
5 - -3; # correct, value is 8
5-(-3); # correct, value is 8
5--3;   # incorrect
```

4. Arithmetic in Perl

- Gotcha 4: be careful with fractional exponents (-4)**(2/3) will be interpreted as exp((2/3) ln(-4)) which is an error since ln(-4) doesn't exist

```
( (-4)**2 )**(1/3); # correct
(-4)**(2/3);        # incorrect
```

5. Named functions in Perl

- Trig functions are in radians: sin(2); asin(1/2);
- Square root: sqrt(9); There is no named cube root function
- Natural exponential: exp(2);
- Natural logarithm: ln(2); log(2); # so ln(x) = log(x) in Perl!!!!
- Base 10 log: logten(2);
- Absolute value: abs(-2);
- Sign / signum function:
  sgn(-2); # returns -1
  sgn(0); # returns 0
  sgn(3.14); # returns 1

6. Relational and logical operators in Perl

- Test whether two numbers are equal:

```
3 == 4; # returns 0 (false)
```

- Test whether two numbers are not equal:

```
3 != 4; # returns 1 (true)
```

- Test using inequalities <, <=, >, >=:

```
3 >= 4; # returns 0
```

6. Relational and logical operators in Perl

- Test whether two strings are equal:
  `"Roy" eq "James"; # returns 0`
- Test whether two strings are not equal:
  `"Roy" ne "James"; # returns 1`

6. Relational and logical operators in Perl

- Are both things true? The and operator &&:

```
(3==(4-1)) && (3==(2+1));
# returns 1
```

- Is at least one thing true? The or operator ||:

```
(3==5) || (3 != 4); # returns 1
```

7. Conditional statements

- If-then statements:
  $a = 5;
  if ($a==4) { $b = 3; }
- The test statement is in rounded parens: ( )
- The code to be executed is in curly braces: { }
- Notice $b=3; is complete, so the end is } not };

7. Conditional statements

- If-then-else statements:

```
$a = 7;
if ($a==7) {
     $b=3;
} else {
     $b=2;
}
```

7. Conditional statements

- If-then-elsif-else:

```
$i = 5;
if ($i == 5) {
     $a = 1;
} elsif (“Roy” eq “James”) {
     $a = 2;
} elsif ($i != 5) {
     $a = 3;
} else {
     $a = 4;
}
```

8. Loops

- For loops:

```
$n = 4;
for ($i=1; $i < 5; $i++) {
      $n = $n + $i;
}
```

- Notice: the recursive assignment $n = $n + $i; is allowed in Perl. We could have also done $n += $i; in place of $n = $n + $i;
- The final value for $n will be 14.

8. Loops

- Foreach loops run through arrays:

```
@evens = ();  # an empty array
foreach my $i (0..50) {
        $evens[$i] = 2 * $i;
}
```

- This will produce an array of 51 even numbers 0, 2, 4,…, 100
- Notice we used $evens[$i], not @evens[$i]

8. Loops

- do-until loop:

```
$a = 3;
do { $a=$a+1; } until ($a==10);
```

- Notice the { } for the code to be executed
- Notice the ( ) for the condition to be tested

PG and MathObjects

1. History

- The PG (Problem Generation) language was written by Michael Gage and Arnold Pizer (U. of Rochester)
- PG is built on Perl
- PG provides macros (prewritten, re-usable code)
- PG displays questions in two modes: HTML and PDF output

1. History

- MathObjects is an extension of PG written by Davide Cervone (Union College)
- M.O. “corrects” some quirks of Perl
- M.O. make writing problems easier
- M.O. provides more macros that are very advanced
- M.O. answer checkers provide more feedback

2. Structure of a PG file

- Tagging info (for the indexing in the National Problem Library)
- Initialization (loading macros, etc.)
- Setup (define parameters, randomization, etc.)
- Main text (the part that gets displayed to students)
- Answer evaluation (checking the submitted answers)
- Solution (optional) and end document (mandatory)

2. Structure of a PG file

- Tagging info:

```
## DESCRIPTION
## Equations for lines
## ENDDESCRIPTION

## KEYWORDS('algebra','line','equation for line')

## DBsubject('Algebra')
## DBchapter('Basic Algebra')
## DBsection('Lines')
## Date('05/26/2011')
## Author('Paul Pearson')
## Institution('Fort Lewis College')
## TitleText1('Intermediate Algebra')
## EditionText1('3')
## AuthorText1('Dewey, Cheatham, and Howe')
## Section1('2.4')
## Problem1('14')
```

2. Structure of a PG file

- Initialization

```
##########################################
#  Initialization

DOCUMENT();
loadMacros(
"PGstandard.pl",
"MathObjects.pl",
"AnswerFormatHelp.pl",
);

TEXT(beginproblem());
```

2. Structure of a PG file

- Setup

```
##############################
#   Setup

Context("Numeric");

$a = non_zero_random(-5,5,1);
$b = random(2,9,1);
```

2. Structure of a PG file

- Main text

```
##########################################
#  Main text

Context()->texStrings;
BEGIN_TEXT
Find an equation for a line through the point
\( ($a,$b) \) and the origin.
$BR
$BR
\( y = \) \{ ans_rule(20) \}
\{ AnswerFormatHelp("formulas") \}
END_TEXT
Context()->normalStrings;
```

2. Structure of a PG file

- Answer evaluation

```
##########################################
#  Answer evaluation

$showPartialCorrectAnswers = 1;

ANS(Formula("($b/$a) x")->cmp());

COMMENT('MathObject version');

ENDDOCUMENT();
```

2. Structure of a PG file

- Comments on Tagging info:
  DBsubject, DBchapter, DBsection are all required to file a problem in the NPL
- Comments on Initialization:
  PGstandard.pl and MathObjects.pl should always be loaded
  TEXT(beginproblem()); dynamically generates the problem number in the homework set

2. Structure of a PG file

- Comments on Setup:
  Don’t over randomize --- choose parameter values that you would like to do by hand when a student brings a question to you

2. Structure of a PG file

- Comments on Main Text:
- A BEGIN_TEXT / END_TEXT block enters a new mode with Perl mode outside, and TEXT mode inside
- In TEXT mode, you can temporarily switch to LaTeX mode via \(\) for inline math, or \[\] for displaystyle math (put on a new line & centered)

```
BEGIN_TEXT
This is inline \( \displaystyle
\left( \frac{3}{4} \right)^2 \).
This is on its own line \[
\frac{3}{4}. \]
END_TEXT
```

2. Structure of a PG file

- Comments on Main Text:
- Inside TEXT mode, you can also switch to Perl mode by using \{ \}, for example

```
BEGIN_TEXT
\{ ans_rule(20) \}
END_TEXT
```

switches into Perl mode and runs the method for generating an answer blank 20 characters wide

2. Structure of a PG file

- Comments on Answer Evaluation:
- The method ->cmp() is defined for any MathObject
- Formula("($b/$a) x")->cmp() takes the student answer and compares it to the Formula object, and returns either 0 or 1
- ANS( ); takes that result and records it in the database of student scores

2. Structure of a PG file

- Comments on Answer Evaluation:
- The COMMENT(‘MathObject version’); only shows up for professors in the Library Browser
- Don’t forget ENDDOCUMENT();

3. Intro to MathObjects

- In Perl,
  `$f = "sin(x)";`
  is just a string
- In MathObjects
  `Formula("sin(x)");`
  is much more than just a string

3. Intro to MathObjects

- A MathObject has methods defined on it
- A method to evaluate functions ->eval()

```
$f = Formula("sin(x)");
$f->eval(x=>5);
```

- A method for (partial) differentiation ->D()

```
$fp = $f->D('x');
```

- A rudimentary simplification method ->reduce()

```
Formula("sin(x) + -4")->reduce(); # sin(x)-4
```

- A method that produces TeX ouput ->TeX()

```
BEGIN_TEXT
What is the derivative of \( $f->TeX() \)
END_TEXT
```

- An answer checker method ->cmp()

```
ANS( $f->cmp() );
```

3. Intro to MathObjects

- Contexts can be modified:

```
Context("Numeric");
$f = Formula("sin(x^2)");

Context()->texStrings;
BEGIN_TEXT
Find the derivative of \( $f \).
$BR
\{ ans_rule(20) \}
END_TEXT
Context()->normalStrings;
```

- Notice sin(x^2) with ^ instead of ** is OK within a MathObject
- Since we changed to texStrings, $f will be interpreted as $f->TeX, and produce the string "\sin(x^2)"
- Notice that we changed back to normalStrings before doing any answer evaluation

3. Intro to MathObjects

- Contexts can be modified:

```
Context("Numeric")->variables->add(
    y=>"Real"
);

$f = Formula("x^2+y^2");

Context("Numeric");
Context()->variables->are(t=>"Real");

$g = Formula("sin(t+pi)");
```

3. Intro to MathObjects

- Contexts can be modified:

```
Context("Numeric");
Context()->operators->undefine("^","**");
Context()->functions->disable("Trig");
Context()->functions->disable("exp");

$f = Formula("x^2");     # error
$g = Formula("sin(x)"); # error
```

- This also disables operators and functions for student answers

3. Intro to MathObjects

- Contexts can be modified

```
Context("Numeric");
Context()->variables->set(
   x => { limits=>[2,5] }
);

$g = Compute("sqrt(x-1)");
```

- Setting limits to [2,5], Webwork randomly selects points x in this interval and compares the values of $g to the values of the student's function at these points (i.e., answer checking is numerical comparison). The default is [-1,1].

3. Intro to MathObjects

- Contexts don't have to be modified

```
Context("Numeric");

$f = Compute("sqrt(x)");
$f->{limits} = [2,5]; # domain issues

$g = Compute("e^(20x)");
$g->{limits} = [-0.25,0.25]; # e^(20) is too large

$h = Compute("ln(x)");
$h->{limits} = [4,10]; # domain issues
```

- Different functions above have different problems that need to be dealt with individually, so don't modify the context (all of them simultaneously)

Resources

Resources

- http://webwork.maa.org/wiki/File:WeBWorK_Problem_Authoring_Tutorial.pdf
- http://webwork.maa.org/wiki/SubjectAreaTemplates
- http://webwork.maa.org/wiki/IndexOfProblemTechniques
- http://webwork.maa.org/pod/pg_TRUNK/
- http://webwork.maa.org/viewvc/system/trunk/pg/macros/
- http://tobi.oetiker.ch/lshort/lshort.pdf