WeBWorK Newbie Guide - version 1.7

Thomas R. Shemanske
Dartmouth College

July 12, 2002

Contents

Chapter 1

Introduction

This document is a beginner's guide to writing and debugging problems in WeBWorK's pg-language. More extensive and higher-level references are available in the technical documentation section of the WeBWorK distribution. Also a great source of help is available via the discussion group on Rochester's site.

The first Newbie Guide was written as a primer for a colleague who decided to jump into the WeBWorK fray. I was starting from scratch as well, and I chose to include a collection of problems the format of which was fairly representative of the problems he would have to design for his course. At the time WeBWorK was in version 1.4, and since then, a large number of things have changed. This seemed a good time to make a number of updates to the document, and to include a few goodies which I have picked up along the way.

You should understand that each pg-problem that you write is essentially a Perl script in which you are producing text which can be translated into LaTeX, HTML, or text. Familiarity with LaTeXwill be a distinct advantage, as will some basic programming experience, although there are enough sample problems in the distribution, that one can make only a few line adjustments to in order to render most simple problems.

Chapter 2

Perl Basics

Perl is short for "Practical Extraction and Report Language", although it has also been called a "Pathologically Eclectic Rubbish Lister". Both are endorsed by Larry Wall, Perl's creator.

2.1 Is Perl difficult to learn?

The question here is a little like the question "Is Math difficult to learn?" I suppose the answer depends a little on what you are trying to accomplish. There is no question that Perl is an extremely powerful (and pervasive) scripting language, and generally what you as a WeBWorK user need to know of it is quite small. On the other hand, Perl is well-known for giving meaning to virtually any random (short) collection of keystrokes one can type on a keyboard. Before getting serious, consider two amusing quotes regarding Perl:

The first is an amusing quote extracted from a Perl FAQ.

Is Perl difficult to learn?

> No, Perl is easy to start learning – and easy to keep learning. It looks like most programming languages you're likely to have experience with, so if you've ever written an C program, an awk script, a shell script, or even BASIC program, you're already part way there.
>
> ...
>
> Things that make Perl easier to learn: Unix experience, almost any kind of programming experience, an understanding of regular expressions, and the ability to understand other people's code. ...

The second quote is joke published by BBspot.

> **Test Shows 99.99% of High School Seniors Can't Read Perl**
>
> San Francisco, CA - Recent results from the standardized Perl Fluency Test showed that 99.99% of US high school seniors can't read Perl. This disturbing

statistic shows that American students are painfully unprepared for life after graduation.

"This shows that there is a real need for a Perl Monk in every classroom," said Perl Monk Kelly Adrity. "We've got computers in every classroom, now we need our kids to be able to use them, and what better way to learn about computers than to learn how to read and write in Perl. I'm glad the budget proposed by President Bush sets aside millions for Perl Monks. America will lead the way in Perl literacy."

The four hour test had 2 sections, a simple translation section and a project section. The first part asked students to translate easy Perl phrases into their standard English equivalent, and the second section required students to produce a simple MP3 player in Perl. "I didn't know what the hell any of it meant," said one Senior, "it had lots of slashes and periods and brackets. It was so confusing. I'm feeling rather nauseous."

Perl experts were astounded by the results. "I was amazed that none of the students were able to read this simple sentence:

```
$_='while(read+STDIN,$_,2048){$a=29;$c=142;if((@a=unx"C*",$_)
[20]&48){$h=5;$_=unxb24,join"",@b=map{xB8,unxb8,chr($_^$a[--$h+84])}
@ARGV;s/...$/1$&/;$d=unxV,xb25,$_;$b=73;$e=256|(ord$b[4])<<9|ord$b[3];
$d=$d>>8^($f=($t=255)&($d>>12^$d>>4^$d^$d/8))<<17,$e=$e>>8^($t&
($g=($q=$e>>14&7^$e)^$q*8^$q<<6))<<9,$_=(map{$_%16or$t^=$c^=
($m=(11,10,116,100,11,122,20,100)[$_/16%8])&110;$t^=(72,@z=(64,72,$a^
=12*($_%16-2?0:$m&17)),$b^=$_%64?12:0,@z)[$_%8]}(16..271))[$_]^
(($h>>=8)+=$f+(~$g&;$t))for@a[128..$#a]}print+x"C*",@a}';s/x/pack+/g;eval
```

I mean, come on, that's so easy," said Paul Chen, Chairman of the Learn Perl or Die Association, which administered the test nationwide. "Teachers need to start with simple phrases like $RF= tr/A-Z/a-z/; and work up from there. We really need to start teaching this in first grade if kids are ever going to understand this by high school."

Not everyone shared Mr. Chen's view about the necessity of adding Perl to early elementary curricula. Programmers Against Perl (PAP) spokesperson, Keith Willingham said, "There's no better way to scare students away from computers than exposing them to Perl." Even experienced programmers are frightened and confused by it. The Perl lobby is just getting too powerful, and they need to be stopped."

Okay, enough with the problems with education in America; we'd better get down to some Perl basics.

2.2 Style

It has to be said. When you write a `pg`-language problem either pretend you are writing it for a colleague who knows absolutely nothing, or for yourself realizing that any cleverness and insight you may have at the moment will have vanished in a month.

That means several things. Annotate your problem with comments (lines beginning with a #). Say useful things, like "don't change the range of values of these variables, roundoff error will drive you mad", or the variable `$number_of_terms` refers to the number of terms in the partial sum of the series.

Use variable names which describe themselves. I hate reading programs with variables `$x1, $x2, $xx1` and so on. Use descriptive variables like `$length_of_ladder` or `$distance_from_wall`. It's unlikely the extra typing will give you carpal tunnel syndrome, and everyone will appreciate the effort, especially you next term when you find a bug in your problem and have to fix it.

2.3 Variables and Arrays

Fundamental to much of what you will do with Perl in WeBWorK involves variables and arrays. All variable names are prefixed with a `$`. Thus `$x`, `$y`, and `$z` are variables while `x`,`y`, and `z` are not, *at least usually.* More about the exceptions in the `pg`-language (graphing and answer macros) later. For example,

```
$x = 12;
$y = ``Hi'';
```

are simple assignment statements in Perl. Note that every Perl statement must end with a semicolon (`;`).

Array names start with an `@`, thus `@array_name` is an array. Arrays can be initialized in several ways. Two common ways are:

- `@fruit = (``apple'', ``banana'', ``orange'', ``pear'');`
- `@fruit = qw(apple banana orange pear);`

Each produces the same array (list) of four items: the character strings `` ``apple'' ``, `` ``banana'' ``, `` ``orange'' ``, and `` ``pear'' ``. The first way is clear, and the second somewhat more convenient using the *white space* between words to delimit the elements of the array. Of course if one of the entries was to be `` ``ripe bananas'' ``, you would have to use the first method, or you would produce an array of five elements.

The elements of the array are referred to as the zeroth, first, second, etc. Symbolically, one writes `$fruit[0]`, `$fruit[1]`, and so on. Note the switch from the `@` to the `$`.

Arrays can also be specified in rather different ways:

- `@integers = (1 ..  7);` [produces (1,2,3,4,5,6,7)]
- `@letters = (B ..  F);` [produces (B,C,D,E,F)]

- `@rational_numbers = (1.5, 4.5 .. 7.5);` [produces (1.5, 4.5, 5.5. 6.5, 7.5)]
- `@real_numbers = (3.14159, @integers);` [produces (3.14159, 1, 2, 3, 4, 5, 6, 7)]

The last example may be somewhat surprising. One might realistically expect that `@real_numbers` is an array with two elements, that latter being an array of seven integers, but no, that is not how Perl works.

Another commonly used technique is to take a *slice* of an array. Suppose `@ALPHABET` is the array `(A, B, ..., Z)` (or `(A .. Z)` using the `..` construction). Also, suppose that `@slice` is the array `(0, 2, 3, 7)`.

Then `@ALPHABET[@slice]` is the array `(A, C, D, H)`, consisting of the 0th, 2nd, 3rd, and 7th elements of the array `@ALPHABET`. This is a very handy construct for manipulating problems in WeBWorK.

2.4 Conditional statements and loops

There will probably come a time when your `pg`-language problem will require cases be handled or answers be computed via iterative methods. Here are a few basics to get you going.

A few samples should allow you to infer the syntax:

```
## Finding the roots of the quadratic $a x^2 + $b x + c = 0
...

if ( $b*$b - 4*$a*$c < 0 )
   {
# Here goes a block of statements handling a negative discriminant
   }
elsif ($b*$b - 4*$a*$c == 0 )
   {
# Block of statements handling double real root
# Note the elsif statement in contrast to elseif or elif from other languages
# Note the use of the == for testing equality unlike the = for assignment
# Replace == with != for not equal
   }
else
  {
#Block of statements handling distinct real roots
  }
```

You can add multiple conditions if need be:

```
if ( condition_1 || condition_2 || or condition_3 )
   {
# Use the || for ‘‘or’’
# Use && instead of the || for ‘‘and’’
   }
```

A simple loop construction:

```
## A simple loop

## Compute partial sums of the harmonic series
## $partial_sum = 0;
## $number_of_terms = 20;

for ( $n = 1; $n <= $number_of_terms; $n++ )
    {
     $partial_sum = $partial_sum + 1/$n;
    }

## Continue processing
```

The syntax `for ( $n = 1; $n <= $number_of_terms; $n++ )` says that for all statements between the braces, execute the statement first with `$n = 1`, then increment `$n` by one (`$n++`), then repeat until you have processed the block with `$n = $number_of_terms`.

2.5 Function Invocation

On a somewhat random note, (many) functions can be invoked in two ways. Surely there is a subtle difference between them, but I have not yet been bitten by it. For example, present in old problems you will find the pairs of statements `&DOCUMENT;` and `&ENDDOCUMENT;`. These can also be written (and this is the preferred invocation) as `DOCUMENT();` and `ENDDOCUMENT();`.

Chapter 3

WeBWorK examples

3.1 Set Definition Files

A set definition file controls which problems are displayed to the students, the order in which the problems appear, and govern the dates during which students are allowed to work on a given set. The name of a set definition file must be of the form setNameOfSet.def which will result in a problem set called "NameOfSet". Typical names might be set2.def or setExam3.def. The typical core of a set definition file has the form:

3.2 The condensed version

```
setNumber=WeBWorKIntro
openDate = 1/7/00 at 6:00am
dueDate = 12/20/01 at 6:00am
answerDate = 12/21/01 at 6:00am
paperHeaderFile = setWeBWorKIntro/paperHeader.pg
screenHeaderFile = setWeBWorKIntro/screenHeader.pg
problemList =
setWeBWorKIntro/webwork_intro_1.pg,           1
setWeBWorKIntro/webwork_intro_2.pg,          1
setWeBWorKIntro/webwork_intro_3.pg,           1
setWeBWorKIntro/webwork_intro_4.pg,           1
setWeBWorKIntro/webwork_intro_5.pg,           1
setWeBWorKIntro/webwork_intro_6/webwork_intro_6.pg,     1
setWeBWorKIntro/webwork_intro_7.pg,           1
```

3.3 The annotated version

```
#######################################
##
## setNumber=NameOfSet
```

```
##
## This is the name of your problem set.
##
## Note:  Do not put any special characters in the name of your set.
## In particular, you may want to avoid things which would make TeX think
## that you are in math mode (like an underscore).  Actually, you can
## use underscores as long as you are careful. For example, use the
## following code in the paperSetHeader.pg file:
## \noindent WeBWorK assignment number \{ protect_underbar($setNumber) \} due
## $formattedDueDate.
##
##
## The ``setNumber='' line is now obsolete, and so can be commented out.
## All problem set names are now derived from the name of the set definition
## file.  See /webwork_system_html/docs/techdescription/setdefinitionformat.html
## for more information.  On the other hand, it is a nice way of
## keeping track of what file you are editing.
##
## When generating the sets for a course from a given textbook, it
## might be reasonable to name sets like setCh1S2 for Chapter 1,
## section 2. For archival purposes, names like
## setIntegrationByParts is clearly more useful .... duh!
##
## Note that while sets shouldn't have underscores in their names,
## the problems names may. More precisely WeBWorK's built in routines which
## display problem names employ protect_underbar.  If you display the names
## yourself, you need to be careful if they contain underscores.
##
########################################


########################################
##
openDate = 1/7/00 at 6:00am
## The date and time at which problems can be viewed and answered
## Professors can view and answer problems before the open date
##
########################################


########################################
##
dueDate = 12/20/01 at 6:00am
## The date and time after which answers will not be recorded
```

```
##
########################################


########################################
##
answerDate = 12/21/01 at 6:00am
## The date and time after which students can view answers and also
## detailed solutions if solutions to problems are provided.
## Professors can view answers and solutions at any time
##
########################################


########################################
##
paperHeaderFile = setWeBWorKIntro/paperHeader.pg
screenHeaderFile = setWeBWorKIntro/screenHeader.pg
##
## Special Header files for what students see on the web
## and on printed versions of the homework
##
## If there is nothing special needed, leave these the lines
## ``paperHeaderFile = ...'' and ``screenHeaderFile = ''
## commented out.  In that case default files will be used. Usually
## these are called ``paperSetHeader.pg'' and ``screenSetHeader.pg''
## and are contained in the templates/ directory. Otherwise, edit the
## files as you like and uncomment the lines.
##
########################################


########################################
##
## Here are the problems in the problem set.
## The order in which they appear in the list below
## governs the Problem number which the students sees
## on the web page (e.g., Problem 1, Problem 2, etc.).
##
## The format for each of the lines is
## problem_file_name , value (i.e., number of points) , number of
## attempts allowed (unlimited if blank).
##
## Each of the problems below is worth 1 point, and students are
```

```
## allowed an unlimited number of tries
##
## See the documentation mentioned above for more detail
##
problemList =
setWeBWorKIntro/webwork_intro_1.pg,   1
## This first problem demonstrates blah, blah,...


setWeBWorKIntro/webwork_intro_3.pg,   1
## This problem demonstrates blah, blah,...

setWeBWorKIntro/webwork_intro_4.pg,   1
## This problem demonstrates blah, blah,...

setWeBWorKIntro/webwork_intro_5.pg,   1
## This problem demonstrates blah, blah,...

setWeBWorKIntro/webwork_intro_6/webwork_intro_6.pg,   1
## This problem demonstrates blah, blah,...

setWeBWorKIntro/webwork_intro_7.pg,   1
## This last problem demonstrates blah, blah,...
##
## End of the problem set
##
#######################################
```

Chapter 4

A problem Template and Description

4.1 The Basic PG-problem Template

```
## -*- perl -*- ##
## Line above puts emacs in perl mode (I like emacs as an editor)
##
## Description
##   This is a generic problem template
## EndDescription
##
## KEYWORDS('Sample','Template')



DOCUMENT();

# Load whatever macros you need for the problem
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl",
           "PGauxiliaryFunctions.pl",
           "PGgraphmacros.pl",
# Uncomment the line below to include some locally-defined macros
#           "Dartmouthmacros.pl"
          );

HEADER_TEXT(...);    # (optional) used only for inserting javaScript into problems.

## Do NOT show partial correct answers
$showPartialCorrectAnswers = 0;
```

```
## Lots of set up goes here


## Ok, we are ready to begin the problem...
##
TEXT(beginproblem());

BEGIN_TEXT
$BR

## In here you will pose the question and provide an answer blank via
## some form of ANS_RULE()

$PAR
END_TEXT

# This is where you will put the correct answers to check against
# those submitted by the student
ANS( answer_evalutors );

ENDDOCUMENT();
```

4.2 The Description

The initial lines are simply comments. All comments in Perl begin with a `#`.

The lines `DOCUMENT();` ...`ENDDOCUMENT();` which bracket everything in the problem appear to simply be delimiters, but actually serves an important purpose. From `.../webwork/system/courseScripts/PG.pl` we read,

> As described in the synopsis, this file and the macros `DOCUMENT()` and `ENDDOCUMENT()` determine the interface between problems written in the PG language and the rest of WeBWorK, in particular the subroutine `createPGtext()` in the file `translate.pl`.
>
> `DOCUMENT()` must be the first statement in each problem template. It initializes variables, in particular all of the contents of the environment variable become defined in the problem environment.
>
> `ENDDOCUMENT()` must the last executable statement in any problem template. It returns the rendered problem, answer evaluators and other flags to the rest of WeBWorK, specifically to the routine `createPGtext()` defined in `translate.pl`.
>
> The `HEADER_TEXT()`, `TEXT()`, and `ANS()` functions load the header text string, the problem text string, and the answer evaluator queue respectively.

Next we have the `loadMacros()` function. Its purpose is to make available all of the functions and libraries that have been written for WeBWorK. I usually keep the list the same even if I am not using all the macros alluded to (e.g. PGgraphmacros.pl). Also, if you have some local macros, they can be included as well (e.g. Dartmouthmacros.pl). Keep those with the others in `/webwork/system/courseScripts/`. The option for local macros referred to above really is intended for locally written macros which you want to be available to all courses using WeBWorK. If you are really writing local macros for your course alone, you can put them in the `macros` subdirectory of your `templates` directory, and then uncomment the `$macroDirectory = "${templateDirectory}macros/";` in your `webworkCourse.ph` file.

Next we set the variable `$showPartialCorrectAnswers = 0;`. This means that if a student gets some of the answers correct on a multi-part problem, no indication will be given as to which ones are correct. If you want to provide provide that information, set the variable equal to 1. Usually you will have `$showPartialCorrectAnswers = 0;` for True/False and Multiple Choice type questions and have `$showPartialCorrectAnswers = 1;` for more complicated of questions. Remember, unless you have implemented an elaborate hints system, it can be quite hard for a student to track down an error especially on a question with multiple answers. The student has no clue whether the error is syntactical or conceptual. Giving some hints by acknowledging partial correct answers can be an important factor in how students feel about WeBWorK.

Now comes the behind-the-scenes set up. Perl statements appear here, perhaps defining random parameters for your problem, defining functions you may want or need, and so on.

Finally, we come to the part of the problem the students will see indicated by the line `TEXT(beginproblem());`. This rather innocuous looking statement also does a great deal. From `PGbasicmacros.pl` we find, `beginproblem()` generates text listing number and the point value of the problem. It will also print the file name containing the problem for users listed in the PRINT_FILE_NAMES_FOR PG_environment variable (in `.../webwork/courses/course_XYZ/webworkCourse.ph`).

The problem which is to be displayed to the student is entered in a `text` environment. Many kinds of things are entered here. Perhaps you have defined a quadratic of the form `x^2 + $b * x + $c` for some "randomly" chosen variables `$b` and `$c`. These will have to be evaluated to specific numbers before generating the HTML code which the browser will use to display the problem to the student. Alternatively, perhaps you have defined an integral using relatively generic LaTeX conventions:

```
\int_{$a}^{$b} f(x) \, dx
```

This too will need to be evaluated (and massaged) to generate acceptable HTML or LaTeX code. In any event, all of this code must be passed to an evaluation function. You will see this accomplished in several (not necessarily interchangeable) ways.

The standard construct is

```
BEGIN_TEXT

*** your problem here ***

END_TEXT
```

This passes all the statements between the `BEGIN_TEXT` and `END_TEXT` through the evaluation routine `EV3` described in more detail than you want in `PGbasicmacros.pl`. Be a little careful if you need to use quotes in a problem. They can lead to unexpected results. You are better off using `$LQ` (left double quote) and `$RQ` (right double quote), which will be interpreted correctly.

Other variations of the `BEGIN_TEXT` ... `END_TEXT` construction which you may see in sample problems are:

```
TEXT(EV2(<<EOT));

*** your problem here ***

EOT
```

or

```
TEXT(EV3(<<'EOT'));

*** your problem here ***

EOT
```

`EV2` is an older version of the evaluator than `EV3`, and behaves slightly differently. Again see `PGbasicmacros.pl` if so inclined. The `BEGIN_TEXT` ... `END_TEXT` construct is equivalent to the `TEXT(EV3(<<'EOT'));` construct, but looks friendlier to use. Also note the single quotes surrounding the `EOT` in the `EV3` invocation; they are required so that `\{...\}` statements are interpreted correctly.

The final construct, `TEXT(blah, blah, blah)`, is the simplest, but does no interpretation or evaluation. Nonetheless, it seems to be indispensable for certain problem constructions such as the matching and multiple choice examples below (at least for older versions of WeBWorK).

Chapter 5

Learning to Walk

In this section are a few problems from the `set0` problem set which comes with your distribution, which have been annotated to make clear the intent of all the statements. Only the first of these is given in gory detail. For the others, we simply indicate the relevant portions. Sometimes parts of the problem are included here for their content to be read as well as their syntax explained.

Before giving those problems, we want to indicate how one proceeds from concept to a finished problem.

5.1 The Gestation Period

- Start with a specific problem as a template:
  Find the derivative of $f(x) = -3x^2 + 5x + 17$.
- Think about variants of the problem and of the type of answers you might want:
  The actual derivative $-6x + 5$, multiple choice from provided selections, the value of the derivative at a point, the equation of the tangent line at a point, and so on.
- Code up a problem for this specific function to test the form of your question.
- Now think about generalizations.
  - You could consider $f(x) = ax^2 + bx + c$ for "randomly" chosen values of a, b, c.
  - You might consider selecting f from among a list of previously chosen functions
- Code these generalizations and debug (see the section below). Do yourself a big favor and put lots of comments in your programs, especially if you had to do something remotely clever. It was out of such comments that this Newbie guide was born.
- Finally, if you have made a problem with "random" coefficients, be sure you test the code with specific values which run over the whole range of possible values to make sure everything works (and appears on the screen) right.

5.2 Debugging PG code

This is unfortunately a short section. If you do nonstandard problems, it is easy to make mistakes, and not so easy to find them.

- There are little things to remember. What students may enter as a valid answer is often quite different than what you may write in a `PG`-problem. For example, if you want students enter an answer which is the product to two numbers 3 and 5, they may write `3*5` or `3 * 5` or `3 5` or `(3)(5)`. You must write one of the first two. Another important example is exponentiation. Students can enter `2^3` or `2**3`; Perl wants `2**3`. In fact `2^3` has a meaning, very different that exponentiation, which can drive you batty. Another place to be careful is with negative numbers. See the Gothas section near the end of the document for details. If you learn to code with an eye towards esthetics, you might write `$a - $b` with some space around the variables. If you are prone to compact writing, you might write `$a-$b`. Surely these are the same. Sometimes. For students they are the same, but not necessarily for you in Perl. If `$a = 3` and `$b = -4`, then the first one evaluates nicely to 7, but there are instances in which the second is interpreted as `3--4` where `--` is a decrement operator in Perl

- If the default display mode for your problems is typeset mode (my default), and the PG problem doesn't compile, you will get a cryptic message on your web browser page like

  ```
  Error: /usr/lib/cgi-bin/webwork/system/cgi-scripts/processProblem8.pl

  Can't rename /var/www/webwork_tmp/m13_test/l2h/28-27120/27120output.html
  ```

  This means nothing other than your PG-program didn't compile. Go back to the problem set selection page (you may have to reload the page), and choose the `text` option. Get the problem again, and you will at least get somewhat more specific cryptic messages, some of which can be useful in discovering the source of your error.

- Another technique is to get a hard copy in TEX format. Often this too will reveal where errors lie.

- If you have no luck, divide and conquer, print values of variables to see if things are working the way you expect, and other generic programming tricks for debugging.

5.3 Sample Problem 1

The first problem presented here is a bit more complicated than many in set0, nonetheless, it is worth listing here because it is not only a realistic problem, but also indicates the process one goes through in writing a problem. The process to which I refer is in going from a specific problem (which has few subtleties), to a generalized problem where you must think carefully about the range of values your variables will assume to guarantee that your code handles the cases correctly. Here is what the output will look like:

Prob. List | Next

WeBWorK

Our records show problem 1 of set Newbie has a score of 50% (0.5 points).

(1 pt) **newbie/problem_1.pg**

Does the function $f(x) = (x+1)^{(13/18)}$ have a tangent line at the point $(-1, f(-1))$? (Answer yes or no) NO

The function f is differentiable at $x = 1$. Find the equation of the tangent line to the curve $y = f(x)$ at $(1, f(1))$, and give its equation in point-slope form: $y - y0 = m(x - x0)$ where

m = ,

x0 = , and

y0 = 2^(13/18) .

Note:*You can earn partial credit on this problem.*

☐ Show Correct Answers

Submit Answers | Preview Answers

Display Mode: ○ text ○ formatted-text ◉ typeset

Show Editor

So the problem is simple: essentially, when is $(x-a)^{m/n}$ differentiable at $x = a$ and $x = a + 2$?

Below is the problem as I would write it in a production run. It is annotated with comments, but only those comments which I might want to see to help me figure out my thinking, and possibly debug the problem in response to a student question.

After presenting this problem, I will dissect it, almost statement by statement, offering a behind-the-scenes look.

```
## Description
##   Tests a knowledge of cusps and tangent lines
## EndDescription

DOCUMENT();
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl"
           );


## Do NOT show partial correct answers
$showPartialCorrectAnswers = 0;

## Consider the function f(x) = (x - $a)^{$m/$n} and determine
## whether there is a tangent line when x = $a, $a + 2

## Generate a "random" point x = $a
$a = random(-5,5,1);

## Note that whether the function is differentiable at x = $a depends
## on the exponent. So we take some care to create exponents some of
## which will produce differentiable functions, some that won't.  In
## particular, if $m < $n the function is not differentiable at x =
## $a.  If $m > $n, whether the function is differentiable depends
## on the parity of $n ($n even gives rise to a cusp).

## Generate the numerator of the exponent.  Let's make them all odd,
## so that the parity of the denominator is unaffected by whether
## $m/$n is in lowest terms

$m = random(7,19,2);

## Generate an array of possible denominators, the first four always
## giving rise to differentiable functions; the second five giving
## rise to a cusp at x = $a if the denominator is even.  Note we could
```

```
## have gone a bit lower, but certainly wanted to avoid $m-6 (which
## could give a value of 1 --- boring), and certainly wanted to avoid
## $m-7 which could produce a zero in the denominator.

@denom = ($m-4, $m-3, $m-2, $m-1, $m+1, $m+2, $m+3, $m+4, $m+5);

## Pick a random index to select the denominator.  Recall that array
## indexing starts at 0

$d_index = random(0,8,1);
$n = $denom[$d_index];


## Ok, we are ready to begin...
##
TEXT(beginproblem());

BEGIN_TEXT
$BR

Does the function \( f(x) = (x - $a)^{($m/$n)} \) have a tangent line
at the point \( ($a, f($a))\)?  (Answer yes or no) \{ans_rule(10) \}

$BR
END_TEXT


## If the numerator is smaller than the denominator, not differentible
if ( $m < $n ){
        $ans = "no";}
## Else we know (by our construction), that $m > $n
## If $n is even, there will be a cusp.
elsif ( $n % 2 == 0 ){
    $ans = "no";}
## Otherwise $m > $n and denominator is odd which is fine
else {$ans = "yes";}


ANS(std_str_cmp($ans));

## Now consider $x = $a + 2, where the function will be differentiable
$b = $a + 2;

BEGIN_TEXT
```

```
$BR

The function \( f \) is differentiable at \( x = $b. \) Find the
equation of the tangent line to the curve \( y = f(x) \)
at \( ($b, f($b)) \), and give its equation in point-slope form:
\(y - y0 = m(x - x0)\) where
$BR

m = \{ans_rule(30)\},
$BR

x0 = \{ans_rule(30)\}, and
$BR

y0 = \{ans_rule(30)\}.
$BR
END_TEXT

ANS(std_num_cmp(($m/$n)*(($b - $a)**($m/$n - 1))));
ANS(std_num_cmp($b));
ANS(std_num_cmp(($b - $a)**($m/$n)));

ENDDOCUMENT();
```

Notice that students can still defeat this problem in the sense that they can enter a correct answer which will be marked wrong by WeBWorK. I asked for three parameters x_0, y_0 and m. Of course the intention is that the students enter the value of \$b for x_0 and $f(\$b)$ for the value of y_0, but of course there are infinitely many correct answers where this is not true. Perhaps the problem should be changed to fix x_0 to be the desired value. Then the remaining answers are uniquely determined.

Now we come to the behind-the-scenes look:

1. `$showPartialCorrectAnswers = 0;`

   This corresponds to not revealing which of the answers may be correct. For example, if the students answers correctly two of the four questions, WeBWorK will respond by saying the student has 50% correct. If you set `$showPartialCorrectAnswers = 1;`, WeBWorK will indicate which of the four answers are correct. The more complicated the problem, the more likely I am to allow them to see which answers are correct.

2. `$a = random(-5,5,1);`

   `random(begin, end, incr)` produces a pseudo-random number between <begin> and <end> in increments of <incr>. The default increment is 1, so the statement `$a = random(-5, 5, 1)` can be abbreviated `$a = random(-5, 5);`

3. `@denom = ($m-4, $m-3, $m-2, $m-1, $m+1, $m+2, $m+3, $m+4, $m+5);`

   Having picked the numerator `$m` for our exponent, we now define an array of possible denominators based upon the numerator, and the constraints we commented on in the problem itself.

4. `$d_index = random(0,8,1);`
   `$n = $denom[$d_index];`

   Knowing that our array of denominators has 9 elements (indexed 0, 1, . . . 8), we choose an index at random and assign to the variable `$n` the corresponding element in the array. Note again how the array is designated `@denom`, while entries are referred to as $denom[<index>], that is the leading `@` changes to a `$`.

5. `BEGIN_TEXT ... END_TEXT`

   Before we get going, note that comments are *not* allowed between the `BEGIN_TEXT` and `END_TEXT` delimiters, so place your comments outside. Anything you write in there will be rendered on the screen for students to see. You can have multiple `BEGIN_TEXT` and `END_TEXT` blocks within a program, so this is not a significant limitation.

   Now what appears between this two delimiters is meant to be rendered for the student in a number of ways. In general, this code is interpreted and a LaTeX document is produced. For students who select to receive a hard copy, this document is TeXed and either postscript or pdf output is produced for the student to view and print. Alternatively, the LaTeX document is run through LaTeX2HTML to produce a nice looking web page on which the student actually does the homework problem.

   The point is that what you write in the text blocks will become either LaTeX code or HTML code. Thus, the code you write here must be converted to the appropriate language, and you must learn to use certain generic control sequences to achieve your goals. Thus for example, a paragraph break in LaTeX is denoted `\par`, while in HTML it is denoted <P>. In the pg language it is denoted `$PAR` which gets translated as appropriate. A list of control sequence is found in `PGbasicmacros.pl`. They include (not an exhaustive list here)

- `$PAR` — paragraph break
- `$BR` — line break
- `$LQ, $RQ` — left and right quotes: ‘‘ and ’’
- `\( \)` — begin and end math mode as in `\( f(x) = \sin(x^2) \)`
- `\[ \]` — begin and end display math mode
- `$LTS, $GTS, $LTE, $GTE` — ¡, ¿, ¡=, ¿= (inequalities)
- `$US` — underscore (without this LaTeX wants to be in math mode)
- `$SPACE` — a space (`\ ` in LaTeX, ` ` in HTML)
- `$BBOLD, $EBOLD` — begin and end bold face type
- `$HR` — horizontal rule across page
- `$LBRACE, $RBRACE` — left and right braces: {, }
- `$LB, $RB` — synonymous with `$LBRACE, $RBRACE`

6. `\{ans_rule(10)\}`

   This is clearly a very important construction since it is how you provide a blank into which students type their answers. The length should provide adequate space for the answer you expect. Be mindful: if you type `ans_rule(10)` all that will be displayed is the text ‘‘`ans_rule(10)`’’. You have to bracket the construct with (escaped) braces (`\{ans_rule(10)\}`) so that the evaluation routine (EV3) interprets it as meaning make an entry for an HTML form.

7. The construct `$a % $b` gives the integer remainder of dividing `$a` by `$b`, with the remainder in the interval `[0, $b-1]`

8. `ANS(std_str_cmp($ans));`

   This is the standard answer evaluator that handles strings. It is case insensitive and any white space is treated as a single space. Thus for example, `Apache really rocks` will be treated the same as `APACHE ReAlLY ROCKS`. It is the most common string function invoked. You can peruse `.../system/courseScripts/PGanswermacros.pl` for more details. Also see Chapter 14.

9.
```
ANS(std_num_cmp(($m/$n)*(($b - $a)**($m/$n - 1))));
ANS(std_num_cmp($b));
ANS(std_num_cmp(($b - $a)**($m/$n)));
```

   This is the standard answer evaluator that handles numeric answers. It will compare the answer the student has answered with the one provided and grant that it is correct if it is within .1% of the given answer. You can change the tolerances which WeBWorK will use, which is quite useful if your answers are very large or very small. It will also accept simple functions as part of the answer: For example, if the answer is 1, then `exp(0)` would be an acceptable answer. You can also require answers in a more restrictive form. See `.../system/courseScripts/PGanswermacros.pl` for more details. Also see Chapter 14.

By the way, it is very important that to denote exponentiation you use `**` and not `^`. Students may use either in answering a question, but Perl wants the `**`.

5.4 Excerpts from set0

- Let's examine this snippet of code:

```
BEGIN_TEXT
Now try calculating the sine of 45 degrees ( that's sine of pi over 4
in radians and numerically sin(pi/4) equals \{ 1/sqrt(2) \} or, more
precisely, \( 1/\sqrt{2} \) ). You can enter this as sin(pi/4) , as
sin(3.1415926/4), as 1/sqrt(2), as 2**(-.5), etc.
END_TEXT
```

Aside from the information conveyed we want to examine the two constructs `\{ 1/sqrt(2) \}` and `\( 1/\sqrt{2} \)`. The first is a Perl construct, the later a TeX construct. The `\{...\}` says to evaluate the expression between the braces and print out the result. In this case `\{ 1/sqrt(2) \}` would display something like .70710678. The other construct is pure TeX which would display $1/\sqrt{2}$. Thus `\{ 2 + 3 \}` will display 5, while `\( 2 + 3 \)` will display $2 + 3$.

- You can embed hyperlinks in your problem page via

```
\{ htmlLink(qq!<URL here>!, "<text identifying hyperlink>") \}
```

- Another snippet: Below, note that `${LQ}` is just the `$LQ` variable (i.e. `$var` and `${var}` are always the same variable). We write `${LQ}Preview` because we don't want a space between `‘‘` and `Preview` (and `$LQPreview` wouldn't work — why?).

```
BEGIN_TEXT

You can always try to enter answers and let WeBWorK do the
calculating. WeBWorK will tell you if the problem requires a strict
numerical answer. The way we use WeBWorK in this class there is no
penalty for getting an answer wrong. What counts is that you get the
answer right eventually (before the due date). For complicated
answers, you should use the ${LQ}Preview${RQ} button to check for
syntax errors and also to check that the answer you enter is really
what you think it is.

END_TEXT
```

5.5 Sample Problem 2

Now that we have done a simple problem with string and numerical answers, we consider one in which the responses are functions. This example is an amended excerpt from `set0`.

Exception to the Rule 1: Note that in the construction below with the `ANS` function, the variable `x` is not prefaced by a $.

```
## Notice that the first occurrence of sin x is between \( and \), and will
## be typeset via LaTeX.  The other occurences of trig functions and
## math symbols will be displayed as plain text.
##
BEGIN_TEXT

This problem demonstrates how you enter function answers into
WeBWorK.
$PAR

First enter the function \( \sin\; x \). When entering the
function, you should enter sin(x), but WeBWorK will also accept sin x
or even sinx, though both of these options will eventually get you
into trouble. If you remember your trig identities, sin(x) =
-cos(x+pi/2), and WeBWorK will accept this or any other function equal
to sin(x), e.g. sin(x) + sin(x)**2 + cos(x)**2 - 1
$BR
\{ans_rule(35) \}

END_TEXT

## Here we enter the answer as a function of the variable x (the
## default).  Alternate variables are handled shortly.
## Below we have the simplest invocation of the answer function
## function_cmp.  The line could also appear as one line:
## ANS(function_cmp(‘‘sin(x)’’));

$ans = "sin(x)";
ANS(function_cmp($ans));



BEGIN_TEXT

We said you should enter sin(x) even though WeBWorK will also
accept sin x or even sinx because you are less likely to make a
mistake.  Read this to say you will get burned.  For example, try
```

```
entering sin(2x) without the parentheses and you may be
surprised at what you get.

Use the Preview button to see what you get.

WeBWorK will evaluate functions (such as sin) before doing anything
else, so sin 2x means first apply sin which gives sin(2) and then
mutiple by x.  Try it.
$BR
\{ans_rule(35) \}

END_TEXT


$ans = "sin(2*x)";
ANS(function_cmp($ans));


## Here we play with a function whose independent variable is
## something other than x.
BEGIN_TEXT

Now enter the function \(2\cos t\).  Note this is a function of
\( t\) and not \( x\). Try entering 2cos x and see what happens.  $BR
\{ans_rule(35) \}

END_TEXT

$ans = "2*cos(t)";
ANS(function_cmp($ans,'t'));
```

Remark: The function `function_cmp` actually can take up to 8 arguments. Before going into the details, think about how WeBWorK knows that $\sin x$ is the same as $-\cos(x + \frac{\pi}{2})$. Symbolic checking would be a nightmare, so the convenient solution is compare the values of the real function and the student's proposed answer at several points. If they match, we concede; if not, they try again. The problem is at which points should you test? Obviously, if the domain is unrestricted, the job is easier, but if the domain is constrained, arbitrary points cannot be chosen. That's where all the other options come into play. If not explicitly given, the various parameters have defaults which are defined in the `/webwork/system/Global.pm` file or the individual course `/webwork/courses/course_XYZ/webworkCourse.ph` file.

From `.../webwork/system/courseScripts/PGanswermacros.pl` we have

- function_cmp($correctFunction)
- function_cmp($correctFunction,$var)

- function_cmp($correctFunction,$var,$llimit,$ulimit)
- function_cmp($correctFunction,$var,$llimit,$ulimit,$relpercentTol)
- function_cmp($correctFunction,$var,$llimit,$ulimit,$relpercentTol,$numOfPoints)
- function_cmp($correctFunction,$var,$llimit,$ulimit,$relpercentTol,
  $numOfPoints,$zeroLevel)
- function_cmp($correctFunction,$var,$llimit,$ulimit,$relpercentTol,
  $numOfPoints,$zeroLevel,$zeroLevelTol)

Here the arguments have the following meanings

- $correctFunction must be a string, e.g. `‘‘sin(x+pi/2)’’`.
- $var is also a string, e.g. `‘‘x’’`.
- The $correctFunction must be defined on the interval [$llimit,$ulimit).
- Typically $var defaults to `‘‘x’’`, $llimit defaults to .00000001, $ulimit defaults to 1, $relpercentTol defaults to .1, $numOfPoints defaults to 3, $zeroLevel defaults to 10^{-14}, and $zeroLevelTol defaults to 10^{-12}.

For example, if your function is $\sqrt{-1-x}$, you would have to enter something like: `function_cmp("sqrt(-1-x)","x",-2,-1)`. The student's answer (a function) is evaluated at `$numOfPoints` random points in the half open interval `[$llimit,$ulimit)`. If the result is within `$relpercentTol*$correctFunction` (with values of absolute value less than `$zeroLevel` handled the same way as in `std_num_cmp`) of `$correctFunction` evaluated at the same points, the student's answer is correct. Allowed functions are listed in Mathematical Functions and Symbols:
`http://www.math.dartmouth.edu/webwork_local_html/available_functions.html`.
Syntax, arithmetic, and other errors are reported to the student.

Chapter 6

Matching Problems

In the first section we build a generic matching question using the functions
`sub match_questions_list (@questions)` or
`sub match_questions_list_varbox($length, @questions)`
which are found in `/var/webwork/system/courseScripts/PGchoicemacros.pl`, and used in older versions of WeBWorK. In the second section, we build a matching problem using object-oriented programming techniques implemented in newer versions of WeBWorK.

6.1 For WeBWorK 1.4, 1.5

The format of this problem is remarkably sensitive to changes. After the example, I will be explicit as to where pitfalls seem to lie. The problem itself is reasonably self-documenting, so I will presume you have read the example before trying to make sense of the remarks which follow.

First, take a look at what the output should look like, then the actual code:

(1 pt) **setShemanske/problem_matching_example.pg**

The first problem below is a matching problem using seven questions and corresponding answers.

Matching Problem 1: In the colors of the visible light spectrum, the acronym ROYGBIV is often used.

- **1.** R stands for
- **2.** O stands for
- **3.** Y stands for
- **4.** G stands for
- **5.** B stands for
- **6.** I stands for
- **7.** V stands for

A. Orange
B. Red
C. Green
D. Blue
E. Yellow
F. Indigo
G. Violet

The seond problem below is a matching problem using four of seven possible questions and corresponding answers.

Matching Problem 2: In the colors of the visible light spectrum, the acronym ROYGBIV is often used.

- **1.** G stands for
- **2.** R stands for
- **3.** B stands for
- **4.** I stands for

A. Indigo
B. Blue
C. Red
D. Green

6.2 Code for example 1

```
## -*- perl -*- ##
## Line above puts emacs in perl mode
##
## Description
##   Matching Problem Example
## EndDescription

DOCUMENT();
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl",
           "PGauxiliaryFunctions.pl",
           "PGgraphmacros.pl");

## Do NOT show partial correct answers
$showPartialCorrectAnswers = 0;

## Generate an array of questions
@questions = ("R stands for",
            "O stands for",
            "Y stands for",
            "G stands for",
            "B stands for",
            "I stands for",
            "V stands for"
             );

## Generate a corresponding array of answers
@answers = ("Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet");

## A list consisting of 0, 1, .. , 6 in "random" order.
@permutation = NchooseK(7,7);

## Ok, we are ready to begin...
##

TEXT(beginproblem());

BEGIN_TEXT
$BR
The first problem below is a matching problem
```

```
using seven questions and corresponding answers.
$BR

$BBOLD Matching Problem 1: $EBOLD In the colors of the visible light
spectrum, the acronym ROYGBIV is often used.  $BR
END_TEXT

TEXT(
     match_questions_list(@questions),
     OL(@answers[@permutation])
    );

ANS(str_cmp([@ALPHABET[invert(@permutation)]]));


## Now we will choose a subset of the possible problems to display

## Choose 4 integers from 0, 1, ... , 6
@subset = NchooseK(7,4);

@subset_of_questions = @questions[@subset];
@subset_of_answers = @answers[@subset];
@new_permutation = NchooseK(4,4);


BEGIN_TEXT
$PAR
The second problem below is a matching problem
using four of seven possible questions and corresponding answers.
$BR

$BBOLD Matching Problem 2: $EBOLD In the colors of the visible light
spectrum, the acronym ROYGBIV is often used.  $BR
END_TEXT

TEXT(
     match_questions_list(@subset_of_questions),
     OL(@subset_of_answers[@new_permutation])
    );

ANS(str_cmp([@ALPHABET[invert(@new_permutation)]]));

ENDDOCUMENT();
```

Remarks: A number of remarks are in order here. The heart of the first problem is contained in the lines:

```
TEXT(match_questions_list(@questions),
     OL(@answers[@permutation]));

ANS(str_cmp([@ALPHABET[invert(@permutation)]]));
```

The function `match_questions_list` simply enumerates and lists the questions, providing an answer box in front of each question. If the answers which are to be supplied are longer than two or three characters, you should use the `match_questions_list_varbox` function which takes two arguments, the first of which is the length of the box, and the second the list of questions.

The line `OL(@answers[@permutation])` displays an ordered list of the answers labeled A, B, C, ... but permuted from their original order by the `@permutation`. To verify the answer is correct, one uses `ANS(str_cmp([@ALPHABET[invert(@permutation)]]));`. That is, when the answers are listed on the web page, `$ALPHABET[$i]` is associated with `$answer[`σ`($i)]`, where σ is the permutation. Thus `$ALPHABET[`σ^{-1}`($i)]` is associated with `$answer[$i]`, the correct answer to `$question[$i]`.

Now for the fussing. First observe that the `math_questions_list` construction wants to be in the `TEXT(...);` context, and not passed through EV2 or EV3.

Second, observe that the answer is in a slightly different form than you may have expected. We are using `str_cmp` instead of something like `std_str_cmp`, and the argument is a list indicated by the extra [ ]'s surrounding `@ALPHABET[invert(@permutation)]`.

6.3 For WeBWorK 1.6, 1.7, ...

In this section, we give an example of the newer way of producing matching questions. This method is probably easier to use than the scripts of the previous section, but that's since more of the underlying structure is hidden from the person writing the problem. On the other hand, since the object is to be able to write these problems as efficiently as possible, the user may be willing to tolerate more of a black box. Parties interested in deeper understanding should read the code in `Match.pm` and `Select.pm` in the `courseScripts` directory.

Again, here is what the output should look like:

Matching Problem 1: In the colors of the visible light spectrum, the acronym ROYGBIV is often used.

Here is the list of questions:
- [ ] **1.** R stands for
- [ ] **2.** O stands for
- [ ] **3.** G stands for
- [ ] **4.** V stands for

Here is the (permuted list) of answers

A. Orange
B. Red
C. Green
D. Violet

This problem below is a matching problem using all three possible questions and corresponding answers.

Matching Problem 2: Match the functions to their derivatives

Here is the list of questions:
- [ ] **1.** $\tan^{-1} x$
- [ ] **2.** $\ln(\sqrt{1+x^2})$
- [ ] **3.** $\ln(1+x^2)$

Here is the (permuted list) of answers

A. $\dfrac{x}{1+x^2}$

B. $\dfrac{1}{1+x^2}$

C. $\dfrac{2x}{1+x^2}$

The core of the problem is reduced to six statements. We discuss them briefly here, and then you can refer to the example below for more detail on their use.

```
## First create a new matched list object (and a pointer to it)
$matched_list_ptr = new_match_list();

## Generate the mathched list, by adding question-answer pairs to the list
$matched_list_ptr -> qa (  ... );

## Choose $k of the pairs to present (1 <= $k <= number of pairs)
$matched_list_ptr -> choose($k);

## From within a BEGIN_TEXT .. END_TEXT environment, print the questions
\{ $matched_list_ptr -> print_q \}

## From within a BEGIN_TEXT .. END_TEXT environment, print the answers
\{ $matched_list_ptr -> print_a \}

## Set up the answer string
ANS( str_cmp( $matched_list_ptr->ra_correct_ans ) ) ;
```

6.4 Code for example 2

```
## Set up a matched list by creating a new matched list object.
## $matched_list_ptr points to a new matched list object.
$matched_list_ptr = new_match_list();

## Now assign to the matched list object an array of "question", "answer"
## pairs from which a subset will be selected.
##
$matched_list_ptr -> qa ("R stands for", "Red",
                         "O stands for", "Orange",
                         "Y stands for", "Yellow",
                         "G stands for", "Green",
                         "B stands for", "Blue",
                         "I stands for", "Indigo",
                         "V stands for", "Violet"
                         );

## Choose four of the possible question-answer pairs
$matched_list_ptr -> choose(4);

## Ok, we are ready to begin...
##
```

```
TEXT(beginproblem());


## In the text block below, we will print the
## selection questions with (uncommented of course)
## \{ $matched_list_ptr -> print_q \}

## and corresponding (and permuted) answers with
## Print the corresponding answers
## \{ $matched_list_ptr -> print_a \}

BEGIN_TEXT
$PAR
This problem below is a matching problem
using four of seven possible questions and corresponding answers.
$BR

$BBOLD Matching Problem 1: $EBOLD In the colors of the visible light
spectrum, the acronym ROYGBIV is often used.  $BR

Here is the list of questions:
\{ $matched_list_ptr -> print_q \}
$PAR

Here is the (permuted list) of answers
\{ $matched_list_ptr -> print_a \}

END_TEXT

## Provide the answer string
ANS( str_cmp( $matched_list_ptr->ra_correct_ans ) ) ;

## Now a simple matching example using questions in math mode
## Note the invocation of math mode is inside the (double) quotes
## which define the question-answer pairs
##
$new_matched_list_ptr = new_match_list();

$new_matched_list_ptr -> qa ("\( \tan^{-1}x \)",
                             "\( \displaystyle{\frac{1}{1+x^2}} \)",
                             "\( \ln(1 + x^2) \)",
                             "\( \displaystyle{\frac{2x}{1+x^2}} \)",
                             "\( \ln(\sqrt{1+x^2}) \)",
```

```
                             "\( \displaystyle{\frac{x}{1+x^2}} \)"
                             );
## Choose all three of the possible question-answer pairs
$new_matched_list_ptr -> choose(3);

BEGIN_TEXT
$PAR
This problem below is a matching problem
using all three possible questions and corresponding answers.
$BR

$BBOLD Matching Problem 2: $EBOLD Match the functions to their derivatives
$BR

Here is the list of questions:
\{ $new_matched_list_ptr -> print_q \}
$PAR

Here is the (permuted list) of answers
\{ $new_matched_list_ptr -> print_a \}

END_TEXT

## Provide the answer string
ANS( str_cmp( $new_matched_list_ptr->ra_correct_ans ) ) ;
```

6.5 Example 3

It is important to note that questions need not have unique answers. The problem below asks which functions are increasing or decreasing on a given interval. The matched list is generated as before with a question and answer, but WeBWorK strips off the duplicate answers before listing them.

```
DOCUMENT();
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl",
           "PGauxiliaryFunctions.pl",
           "PGgraphmacros.pl",
           );
```

```
## Do NOT show partial correct answers
$showPartialCorrectAnswers = 0;

## Set up a matched list by creating a new matched list object.
## $matched_list_ptr points to a new matched list object.
$matched_list_ptr = new_match_list();

## Now assign to the matched list object an array of "question", "answer"
## pairs from which a subset will be selected.
##
$matched_list_ptr -> qa ("\( x^2 \)", "Increasing",
                         "\( x^4 \)", "Increasing",
                         "\( e^{-x^2} \)", "Decreasing",
                         "\( \ln(x) \)", "Increasing",
                         "\( \ln(e^{-x^2})) \)", "Decreasing",
                         );

## Choose four of the possible question-answer pairs
$matched_list_ptr -> choose(4);

## Ok, we are ready to begin...
##

TEXT(beginproblem());

BEGIN_TEXT

On the interval \( (0,1) \), determine which of the functions below
are increasing and which are decreasing.

\{ $matched_list_ptr -> print_q \}
$PAR

Here is the (permuted list) of answers
\{ $matched_list_ptr -> print_a \}

END_TEXT

## Provide the answer string
ANS( str_cmp( $matched_list_ptr->ra_correct_ans ) ) ;

ENDDOCUMENT();
```

Chapter 7

Multiple Choice Problems

7.1 For WeBWorK 1.4, 1.5

Here is a simple multiple choice question: identify the interval containing π. In this set up, we introduce a permutation so that even though the answers are fixed, the correct answer will not necessarily be the same for all students.

Here is some sample output:

Previous | Prob. List

WeBWorK

Our records show problem 33 of set Shemanske has not been attempted.

(1 pt) **setShemanske/problem_mult_choice_example.pg**

The real number π is which interval?

A. the interval $[2, 3)$

B. the interval $[3, 4)$

C. the interval $[4, 5)$

D. the interval $[5, 6)$

☐ Show Correct Answer

Submit Answer | Preview Answer *Note: it is after the due date. Answers available.*

Display Mode: ○ text ○ formatted-text ◉ typeset

Show Editor

Logout | Feedback | ? Help | Problem Sets | Enter Professor's Page

```
Problem Set Version Number: 129355
Page produced by script:
/usr/lib/cgi-bin/webwork/system_1.7/cgi-scripts/processProblem8.pl
```

The snippet of code is below. This is a good deal simpler than the matching questions, and hopefully the code is clear. Once again we have to retrieve the answer from an ordered list. The correct answer is `$answers[1]` (the *second* answer!). The letters are listed with `$ALPHABET[i]` corresponding to `$answers[`σ`(i)]`, so `$answers[1]` corresponds to `$ALPHABET[`σ^{-1}`(1)]`.

```
@answers = ("the interval \( [2, 3) \)",
            "the interval \( [3, 4) \)",
            "the interval \( [4, 5) \)",
            "the interval \( [5, 6) \)");

## A list consisting of 0, 1, 2, 3 in "random" order.
@permutation = NchooseK(4,4);

## Ok, we are ready to begin...
##

TEXT(beginproblem());

BEGIN_TEXT
$BR
    The real number \( \pi \) is which interval? \{ans_rule(10)\}
$BR
END_TEXT

TEXT(OL(@answers[@permutation]));

@inverted_alphabet = @ALPHABET[invert(@permutation)];

## The answer is $answers[1]
ANS(std_str_cmp($inverted_alphabet[1]));
```

7.2 For WeBWorK 1.6, 1.7, ...

There are new multiple choice objects which again make this type of problem somewhat easier. Below is some sample output followed by (hopefully) self-documenting code. Note that in the old style the student entered letters corresponding to correct answers; here we just have checkboxes.

1 pt) **setShemanske/problem_multiple_choice_new.pg**

Which integers below are congruent to 1 (mod 4)?

- ☐ **A.** 17
- ☐ **B.** 41
- ☐ **C.** 11
- ☐ **D.** 31
- ☐ **E.** 7
- ☐ **F.** 5
- ☐ **G.** 13
- ☐ **H.** 37
- ☐ **I.** 3
- ☐ **J.** 2
- ☐ **K.** 23
- ☐ **L.** 29
- ☐ **M.** 19
- ☐ **N.** All of the above
- ☐ **O.** None of the above

☐ Show Correct Answer

Note: it is after the due date. Answers available.

Display Mode: ○ text ○ formatted-text ◉ typeset

Logout Feedback ? Help

```
Problem Set Version Number: 154218
Page produced by script:
/usr/lib/cgi-bin/webwork/system_1.7/cgi-scripts/processProblem8.pl
```

```
DOCUMENT();
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl",
           "PGauxiliaryFunctions.pl",
           "PGgraphmacros.pl"
           );


## Do NOT show partial correct answers
$showPartialCorrectAnswers = 0;

## Make a new checkbox multiple choice object
$checkbox_mc = new_checkbox_multiple_choice();

## Insert a question, and all correct answers
$checkbox_mc -> qa("Which integers below are congruent to 1 (mod 4)?",
                   5, 13, 17, 29, 37, 41);

## Insert some bogus answers
$checkbox_mc -> extra(2, 3, 7, 11, 19, 23, 31);

## Insert some answers which will always be at the end of the list
$checkbox_mc -> makeLast("All of the above", "None of the above");

TEXT(beginproblem());


## Print the question and answers
BEGIN_TEXT
$BR
\{ $checkbox_mc -> print_q \}
$PAR
\{ $checkbox_mc -> print_a \}

$PAR
END_TEXT


ANS( checkbox_cmp( $checkbox_mc -> correct_ans ));

ENDDOCUMENT();
```

Chapter 8

True - False

Of course, a true-false set of questions is just a special matching problem such as the third example in the chapter on matching problems, however it is probably more esthetic to enter T and F instead of A and B, so the WeBWorK folks have designed a special set of macros for that as well. We just give a version for the newer versions of WeBWorK.

```
DOCUMENT();
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl",
           "PGauxiliaryFunctions.pl",
           "PGgraphmacros.pl",
           );


## Do NOT show partial correct answers
$showPartialCorrectAnswers = 0;

## Set up a true-false list by creating a new select list object.
## $tf_list_ptr points to a new select list object.
$tf_list_ptr = new_select_list();

## Now assign to the true-false list object an array of
## "question", "answer" pairs from which a subset will be selected.
##
$tf_list_ptr -> qa ("There are infinitely many primes", "T",
                    "Fermat got it right", "T",
                    "Linux rocks", "T",
                    "\( m^2 + m + 41\) is prime for all integers \(m\)", "F",
                    "The coefficients of all cyclotomic polynomials are either 0, 1
                    );
```

```
## Choose four of the possible question-answer pairs
$tf_list_ptr -> choose(4);

## Ok, we are ready to begin...
##

TEXT(beginproblem());

BEGIN_TEXT

Determine whether the statements below are true or false.  Enter
$BBOLD T $EBOLD for true and $BBOLD F $EBOLD for false.

\{ $tf_list_ptr -> print_q \}


END_TEXT

## Provide the answer string
ANS( str_cmp( $tf_list_ptr->ra_correct_ans ) ) ;

ENDDOCUMENT();
```

Chapter 9

Graphs

The standard WeBWorK documentation contains several pages on creating and using graphs
(`http://webhost.math.rochester.edu/webworkdocs/docs/pglanguage/tutorial/`).
This section will supplement and occasionally overlap with that documentation.

9.1 Graph Objects

To draw a graph, you first need a set of axes on which to draw it. You need to specify the domain and codomain, whether you want tick marks on the axes or a grid, where you want the axes to appear, and how large (in pixels) the final image should be.

Here is a simple example. The domain and codomain are often computed in terms of your (variable) function, so I have left them as the variables `$xmin, $xmax, $ymin, $ymax`. The `axes` option puts the axes in the usual place; the `grid` option indicates there will be 12 vertical grid lines and 20 horizontal ones. These too can be given by variables. When graphs are displayed they are usually displayed as a 100×100 thumbnail image (though this can be changed with the `height` and `width` options. In general, this seems to work well since you often have four graphs on a line. On the other hand, the thumbnail often does not display enough detail for a student to be able to answer the question, so for each thumbnail there is the original image which will pop up when you click on the thumbnail. Actually, a new browser window opens (often right on top of your existing one), with only the enlarged image displayed. The default size of this enlargement is 200×200 pixels; I have selected a somewhat larger image specifying 300×300 via the `pixels` option.

```
# Set the domain of f
$xmin = -6;
$xmax = 6;

## Set the visible range
$ymin = -20;
$ymax = 20;
```

```
$graph_object = init_graph($xmin, $ymin, $xmax, $ymax,
                    'axes' => [0,0],
                    'grid' => [12,20],
                    'pixels' => [300,300]
                    );
```

9.2 Adding functions to your graph object

So now that you have an object on which to draw your graph(s), we should specify a function. Now as the most trivial of examples, let us suppose we want to draw the graph of $y = x^2$ on the axes we have just defined. Nothing could be simpler:

```
$f = ``x^2 for x in [$xmin, $xmin] using color:blue and weight:2'';
add_functions($graph_object, $f);
```

Exception to the Rule 2: Note that in this construction the variable `x` is not prefaced by a $.

Note: The function `plot_functions` can be used interchangeably with the function `add_functions` if one seems easier to remember.

Usually, one has somewhat more interesting examples to graph. Suppose that `$b` and `$c` are real numbers between -3 and 3. You want to graph `x^2 + $b x + $c`. Now if b or c is negative, you will obtain an expression like $x^2 + -3x + -2$, less than optimal for parsing. So one uses the `FEQ` function (Format EQuations) and writes

```
$f = FEQ(``x^2 + ${b} * x + ${c} for x in [$xmin, $xmin]
                          using color:blue and weight:2'');
```

which will return the string (assuming the values of `$b, $c` above)

```
$f = ``x^2 - 3 * x - 2 for x in [$xmin, $xmin]
                          using color:blue and weight:2''
```

In particular, FEQ will take care of adjacent `+-` or `-+` or `--` signs with no prompting.

Certainly, you can draw multiple functions with possible different domains on the same graph object. As a simple example, let's draw a piecewise linear function.

```
$f1 = ``-2 * x - 6 for x in [$xmin, -2) using color:blue and weight:2'');
$f2 = ``x for x in [-2, 3) using color:blue and weight:2'');
$f3 = ``x - 4 for x in [3, $xmax] using color:blue and weight:2'');
add_functions($graph_object, $f1, $f2, $f3);
```

The discontinuity at $x = 3$ will be clearly shown with an open circle at $(3, 3)$ and a filled circle at $(3, -1)$.

9.3 Displaying your graphs

The usual way in which you would display a graph is via the statements:

```
BEGIN_TEXT

Below is the graph of a function \( f \):
($BBOLD Click on image for a larger view $EBOLD)
$PAR
\{ image(insertGraph($graph_object)) \}
$PAR

END_TEXT
```

There are two functions which are called here: `image` and `insertGraph`. Often they are used together as above, but occasionally they are invoked separately.

For example, the actual "value" of `$graph_object` is something like `WWPlot=HASH(0x879f278)`. The actual graph image (more precisely, the full path to the image) is provided by `insertGraph($graph_object)`. That is, it returns a path like `.../webwork_tmp/course_XYZ/gif/trs-82786-setShemanskeprob2gif1.gif` The `image` macro generates the appropriate code for insertion of a graphic into an HTML or LaTeX document.

The function `image` takes optional parameters. For example:

```
image($image, width => 100, height => 100, tex_size => 800)
or
@image([$image1, $image2], width => 100, height => 100, tex_size=>800)
```

The width and height parameters are used in generating HTML displays (via LaTeX2HTML), while the `tex_size` parameter is used in LaTeX mode as when generating hardcopy. It is used in the following way: the rendered images size will be `tex_size * .001 * \linewidth`, so `tex_size=>800` corresponds to 80% of the line width (which can of course vary depending on whether you are in one or two column mode.

Finally, note that you can display a row of images (and optional captions) using the format `imageRow([$image1, $image2 ], [$caption1, $caption2])`. More about captions shortly.

We should also note that graphs can be labeled and captioned. The distinction here is that labels appear as a part of the graph object, while captions are separate from the object.

9.4 Labelling your graphs

First we consider an example with labels. We give two methods for doing this. The first method is older, somewhat simpler, but less robust. The second method is newer, more daunting in appearance, but provides features which may be use to you. The code is self-documenting.

```
##Initialize the graph object
##
$graph_object = init_graph($xmin,$ymin,$xmax,$ymax,
                    'axes' => [0,0],
                    'grid' => [12,20],
                    'pixels' => [300,300]
                    );

## Define the function to graph
$f = FEQ("${a}*x^2 + ${b}*x + ${c} for x in [$xmin, $xmax] using
color:blue and weight:2");


## Add the function to the graph object
##
add_functions($graph_object, $f);


## Add the label "My label" at the point (1,5).  The justification
## means that (1,5) will be at the top left of the label.
## Other options are {top, middle, bottom} and {left, right, center}
##
$label_object_reference = new Label (1,5,'My label','blue',('top','left'));

## Add the label to the graph object
$graph_object -> lb($label_object_reference);

## Multiple labels can be added with one \texttt{lb} statement, e.g.,
## $graph_object -> lb($label_1, $label_2, $label_3);

## Ok, we are ready to begin...
##
TEXT(beginproblem());

BEGIN_TEXT

Below is the graph of a function \( f \):
($BBOLD Click on image for a larger view $EBOLD)
$PAR

\{ image(insertGraph($graph_object)) \}
```

```
$PAR

END_TEXT


## Here is a fancier approach which is smart about
## relating the placement of the label to the graph
## of the function

$new_graph_object = init_graph($xmin,$ymin,$xmax,$ymax,
                    'axes' => [0,0],
                    'grid' => [12,20],
                    'pixels' => [300,300]
                    );

## Define the function to graph
$g = FEQ("${a}*x^2 + ${b}*x + ${c} for x in [$xmin, $xmax] using
color:blue and weight:2");


## Add the function to the graph object
##
($g_ref) = add_functions($new_graph_object, $g);

## Now, $g_ref is a pointer to the function g containing information
## about the function.
##
## For example, $g_ref->rule points to the rule by which to
## calculate the value of the function g.
##
## The expression &{$g_ref->rule}(3) calculates the value of the
## function g at the point 3.


## Add the label "My label" at the point ($xcoord, 3+g($xcoord)).
## The justification means the point will be at the top left of the label.
## Other options are {top, middle, bottom} and {left, right, center}
##

$xcoord_label = 1;
$new_label_object_reference =
         new Label ($xcoord_label,3+&{$g_ref->rule}($xcoord_label),
         'My label','blue',('top','left'));
```

```
## Add the label to the graph object
$new_graph_object -> lb($new_label_object_reference);


BEGIN_TEXT

Below is the graph of the same function \( f \):
($BBOLD Click on image for a larger view $EBOLD)
$PAR

\{ image(insertGraph($new_graph_object)) \}

$PAR

END_TEXT
```

9.5 Captioning your graphs

Finally, graphs can be captioned (with descriptive captions or as part of a multiple choice problem) in the following manner. Suppose that `@graphs` is an array of four graph objects. We could write:

```
## Make an array of graph paths corresponding to the graph objects.
##
@graph_paths = ();
for ($i = 0; $i <= 3; $i++)
    {
     $graph_path[$i] = insertGraph($graphs[$i]);
    }


## Make an array of captions for the graphs
@captions = (‘‘Graph 1’’, ‘‘Graph 2’’, ‘‘Graph 3’’, ‘‘Graph 4’’);

## To display the graphs in a row with their captions
## The ~~ escapes the @ sign
##
TEXT(imageRow(~~@graph_path, ~~@captions));
```

Chapter 10

Tables

Table macros use the HTML table formats, or the LaTeX "tabular" environment. The format is fairly straightforward

```
TEXT(
     begintable (number_of_columns),
        row(item_1, item_2, ... item_{number_of_columns}),
        row(@array_of_entries),
     endtable()
    );
```

No checking is made to see if you have entered the correct number of entries in each row. Also note that if `array_of_entries` is a typical row of your table, you can specify the `number_of_columns` as `scalar(array_of_entries) + 1`.

Below is an a rather complicated example which uses tables, graphs, and multiple choice. We use tables to provide information about the first and second derivative of a polynomial function f with critical points at integers $a < b < c$. The values of the first and second derivatives of f are specified at points between the critical points, and a table of those values is displayed. Then four graphs are displayed: $f(x)$, $-f(x)$, $f(-x)$, $-f(-x)$, and the student is told to choose the correct one. We use the `&{$g_ref->rule()}()` construct to produce the values of the first and second derivatives which appear in the table. One can also do this more elegantly via the `FUN.pm` library, see the chapter on functions.

```
## variables $a, $b, $c denote the zeros of the derivative of $f
## $y0 is the value $f(0)
$a = random(-6,-3,1);
$b = random(-2,1,1);
$c = random(2,6,1);
$y0 = random(-2,2,1);

## Set the domain of f
$xmin = -8;
$xmax = 8;
```

```
## Set the visible range
$ymin = -100;
$ymax = 100;

## In order to use the construction  \verb!&{$g_ref->rule()}()!,
## I need a reference to a function.
##
## One way I know how to do this is via add_functions
## You could get this out of the FUN.pm library.  See
## the the chapter on Functions


$null_graph_object = init_graph($xmin,$ymin,$xmax,$ymax,
                    'axes' => [0,0],
                    'grid' => [16,20],
                    'pixels' => [300,300]
                    );

## Define the function $f to graph as well as it's first and second
## derivatives $fp and $fpp
##
$f = FEQ("(x**4)/4 - ($a + $b + $c) * (x**3)/3  +
     (($a)*($b) + ($a + $b) * ($c))* (x**2)/2 -
     ($a)*($b)*($c)*x + $y0 for x in [$xmin, $xmax]
     using color:blue and weight:2");

$fp = FEQ("(x - $a) * (x - $b) * (x - $c)
      for x in [$xmin,$xmax] using color:red and weight:2");

$fpp = FEQ("(x - $a)*(x - $b) + (x - $a)*(x - $c) +
       (x - $b)*(x - $c) for x in [$xmin, $xmax]
       using color:yellow and weight:2");

## Generate the pointers which allow us to compute values of the functions
($f_ref, $fp_ref, $fpp_ref) = add_functions($null_graph_object, $f, $fp, $fpp);


## Build an array of four graph objects
@graph_object = ();
for ($i = 0; $i <= 3; $i++)
    {
     $graph_object[$i]=init_graph($xmin,$ymin,$xmax,$ymax,
                       'axes' => [0,0],
                       'grid' => [16,20],
```

```
                        'pixels' => [300,300]
                        );
    }

$minus_f = FEQ("-(x**4)/4 + ($a + $b + $c) * (x**3)/3  -
          (($a)*($b) + ($a + $b) * ($c))* (x**2)/2 +
          ($a)*($b)*($c)*x - $y0 for x in [$xmin, $xmax] using
          color:blue and weight:2");

$f_of_minus_x = FEQ("(x**4)/4 + ($a + $b + $c) * (x**3)/3  +
                (($a)*($b) + ($a + $b) * ($c))* (x**2)/2 +
                ($a)*($b)*($c)*x + $y0 for x in [$xmin, $xmax] using
                color:blue and weight:2");


$minus_f_of_minus_x = FEQ("-(x**4)/4 - ($a + $b + $c) * (x**3)/3  -
                      (($a)*($b) + ($a + $b) * ($c))* (x**2)/2 -
                      ($a)*($b)*($c)*x - $y0 for x in [$xmin, $xmax] using
                      color:blue and weight:2");

## Add the function and variations to the graph objects
##
add_functions($graph_object[0], $f);
add_functions($graph_object[1], $minus_f);
add_functions($graph_object[2], $f_of_minus_x);
add_functions($graph_object[3], $minus_f_of_minus_x);


## This next part assigns an actual path for a given graph image
## For example the value of $graph[0] is WWPlot=HASH(0x879f278)
## The actualimage is insertGraph($graph[0]), namely
## .../webwork_tmp/m13_test/gif/trs-82786-setShemanskeprob2gif1.gif
##
@graph_path=();
for ($i = 0; $i <= 3; $i++)
    {
      $graph_path[$i] = insertGraph($graph_object[$i]);
    }

## Define an array of captions with the letters A - D
@captions = @ALPHABET[0..3];

## NchooseK(n,k):  choose k integers randomly from 0, 1, ..., (n-1)
## and put them in the array @indices
```

```
@indices = NchooseK(4,4);

## Mix up the graphs
@permuted_graph_path = @graph_path[@indices];


BEGIN_TEXT
Using the information about the first and second
deriviatives provided in the tables below, enter the letter of the
graph below which corresponds to the function.
\{ ans_rule(10) \}
$PAR
END_TEXT


## Build the table of values here
## Test the half-integer points between the critical points
## Should be adequate for the second derivative as well.
##

## The number of columns will be $c - $a + 3
$number_of_columns = $c - $a + 3;
## Now create the array of values at which to evaluate the derivatives
@x_coords = ();

## Let yp_coords be the array of values of the first derivative
## at the specified points
@yp_coords = ();

## Let ypp_coords be the array of values of the second derivative
## at the specified points
@ypp_coords = ();

$x_coords[0] = "\(x\)";
$yp_coords[0] = "\(f'(x)\)";
$ypp_coords[0] = "\(f''(x)\)";

## A little kludgey on the indexing
## $x_coords[1] = $a - .5
## $x_coords[$number of columns] = $c + .5
## yp_coords and ypp_coords get the values of
## the first and second derivatives
##
for ($i = 1; $i < $number_of_columns; $i++)
```

```
    {$x_coords[$i] = $a - 1.5 + $i;
     $yp_coords[$i] = &{$fp_ref->rule}($x_coords[$i]);
     $ypp_coords[$i] = &{$fpp_ref->rule}($x_coords[$i]);
    }


TEXT(
     begintable ($number_of_columns),
       row(@x_coords),
       row(@yp_coords),
       row(@ypp_coords),
     endtable()
    );


## Print the graphs out here
TEXT(imageRow(~~@permuted_graph_path, ~~@captions));

ANS(str_cmp( [@ALPHABET[ invert(@indices)]]));
```

Chapter 11

Functions

Here we talk a little about using functions in WeBWorK, both simply in terms of algebraic constructions, and also in the context of passing an abstract function to a subroutine.

11.1 A Simple Example

Consider a simple function $f(x) = x^2$. We could define f by

```
sub my_f {
    my $x = shift;
    return $x**2;
        }
```

The statement `my $x = shift;` has two parts. The first `my $x` declares `$x` as a local variable, while the assignment `my $x = shift;` assigns to `$x` the first (and only in this case) argument passed to the function.

The function returns the value of `$x` squared. Note the use of `**` as the exponentiation operator. This is pure Perl, so the caret has a completely different meaning.

To print a little table of values of $f(x)$, $g(x)$, and $f(g(x))$ we might write (again the code is reasonably self-documenting):

```
## Define the first function f(x)
sub my_f{
    my $x = shift;
    return $x**2;
       }
## Make a nice version to print
$f_print = FEQ("\(f(x) = x^2 \)");

## Define the second function g(x)
sub my_g{
    my $x = shift;
```

```
    return $x**3;
        }

## Make a nice version to print
$g_print = FEQ("\(g(x) =  x^3 \)");

## Pick some random values at which to evaluate the functions
@x_values = (1, 2, 3, 5, 7);

## Set up some arrays to collect the values
@f_of_x = ();
@g_of_x = ();
@f_of_g_of_x = ();

## Evaluate f(x), g(x), f(g(x)) and add to arrays
## Yes, I know I could be more efficient
##
## scalar(@x_values) is the number of elements in the array @x_values
for ($i = 0; $i < scalar(@x_values); $i++)
{
    $f_of_x[$i] = my_f($x_values[$i]);
    $g_of_x[$i] = my_g($x_values[$i]);
    $f_of_g_of_x[$i] = my_f(my_g($x_values[$i]));
}


## Ok, we are ready to begin the problem...
##
TEXT(beginproblem());

BEGIN_TEXT
$BR
Here is a table of values of functions $f_print and $g_print.
$PAR
END_TEXT

## Print a table of values
TEXT(
     begintable (scalar(@x_values) + 1),
       row(("\( x \)",@x_values)),
       row(("\( f(x) \)",@f_of_x)),
       row(("\( g(x) \)",@g_of_x)),
       row(("\( f(g(x)) \)",@f_of_g_of_x)),
     endtable()
```

```
    );


ENDDOCUMENT();
```

You might notice a little comfort with the use of Perl here. The statement `row(("\( x \)",@x_values))` takes advantage of the way that Perl handles arrays. That is, the function `row` takes an array as an argument. We want an array whose first entry is the label for the row, followed by the actual values which already appear in an array. You will recall from an early section of this guide that if `@a`, `@b`, `@c` are arrays, then the statement `@a = (@b, @c)` creates an array whose members are those of the array `@b` followed by those of the array `@c`.

Certainly, you can define more involved functions.

If `$a`, `$b`, and `$c` are constants already defined in your problem, you could define a (generic) quadratic as:

```
sub my_f{
    my $x = shift;
    return $a * $x**2 + $b * $x + $c;
        }
```

You could be braver and define a generic polynomial function by

```
sub polynomial{
## The function polynomial takes three arguments.  The first is
## the degree, the second an array of (degree+1) coefficients, and
## the third the value of the independent variable.
##
    my $degree = shift;
    my @coefficients = shift;
    my $x = shift;

## Define some local variables
    my $i;
    my $sum = 0;
    for ($i = 0; $i <= $degree; $i++)
        $sum = $sum + $coefficients[$i] * ($x**($i));
    return $sum;
        }
```

Could you have functions of more than one variable? Sure

```
## Define f(x,y) = x^2 + y^2
sub my_f{
    my $x = shift;
    my $y = shift;
    return $x**2 + $y**2;
        }
```

11.2 A More Involved Example

Now suppose that you want to pass a function as an argument to another function. For example, you want to use Euler's method to estimate the value of the solution to the boundary value problem $\frac{dy}{dx} = f(x, y);\ y(x_0) = y_0$.

Here the idea is to define a subroutine which uses the Euler method on an arbitrary function given the starting point, stepsize, and number of iterations to perform.

Here is the core of the Euler subroutine

```
## Usage:  euler(function_reference, $x_0, $y_0, $step_size, $number_of_steps);
## Returns an array (@xcoords, @ycoords)
##
sub euler {
## Grab the parameters which were passed in the function call
    my $fn_ref = shift;
    my $local_x0 = shift;
    my $local_y0 = shift;
    my $local_step_size = shift;
    my $local_number_of_steps = shift;

## a local variable
    my $i;

## Initialize the arrays of coordinates to return
    my @xcoords = ($local_x0);
    my @ycoords = ($local_y0);

## Perform Euler's method the requisite number of times
    for ($i = 1; $i <= $local_number_of_steps; $i++)
    {
        $xcoords[$i] = $xcoords[$i-1] + $local_step_size;
        $ycoords[$i] = $ycoords[$i-1] +
                  $local_step_size * &$fn_ref($xcoords[$i-1], $ycoords[$i-1]);
    }

    return (@xcoords, @ycoords);
}
```

Next we make a function reference to pass to the Euler subroutine

```
## Make a function object for f(x,y) = x^2 + y^2
$f_ref = new Fun( sub {
                        my $x = shift;        ## local variable
                        my $y = shift;        ## local variable
```

```
    return $x**2 + $y**2  ## return f(x,y)
} );
```

Finally, we invoke the Euler routine with the following syntax, the function reference being `$f_ref->rule()`

```
@x_and_y_coords = euler($f_ref->rule(),$x0,$y0,$step_size,$number_of_steps);
```

Note that the array which is returned has length `2*($number_of_steps + 1)` with entries $x_0, x_1, \ldots x_n, y_0, y_1, \ldots, y_n$ where n is `$number_of_steps`

Chapter 12

Idiosyncrasies

The function FEQ (Format EQuations) is a simple parser designed mainly to resolve adjacent sign conflicts (e.g. $2 + -3x$). It does not do any arithmetic.

There are many (at least seemingly) idiosyncratic/outdated constructions in the FEQ routine which have become ingrained in existing problem sets.

12.1 Example

For example, in defining a graph, one might have once written:

```
##
$a = -2;
$b = -2;

## This expression will be correctly parsed by FEQ
$f = FEQ("$a*x^2 ? {$b}*x  for x in [$xmin, $xmax] using color:blue
and weight:2");
```

in an attempt to resolve problems with the sign of $b and the preceding operator.

Now you should just write

```
$f = FEQ("${a}x^2 + ${b}x for x in [$xmin, $xmax] using color:blue
and weight:2");
```

and FEQ will take care of adjacent +- or -+ or -- signs with no prompting.

12.2 Example

As another example, the exponentiation operator is one of which to be very careful. It is fine to use ^ in FEQ or in a TEX math construction, however using ^ instead of ** outside of these constructs (generic Perl statements) will lead to unpredictable results which may take a while to track down.

Chapter 13

Gotchas

Here are a few items which have either bitten me, or about which I have been told to watch out.

13.1 Exponentiation

The exponentiation operator is one of which to be very careful. In all answers which a student provides, there is no difference in whether a caret (`^`) or `**` is used to denote exponentiation. In any `FEQ` statement it is fine to use `^` instead of `**`. Within math mode in TEX it is required to use the caret. However, outside of these contexts, you should always use `**`. This includes generic Perl statements as well as *your* answers (in the `ANS` construct). Generally, the `PG` problem will compile, but the results will be unpredictable.

Also, a standard programming error is being cavalier with non-integral exponents. We all know that $(-2)^{2/3}$ is a perfectly fine real number, however most programming languages (Perl included) will evaluate this via $e^{(2/3)\log(-2)}$ leading to certain difficulties.

When using variables for exponents in a TEX expression (i.e within `\( ... \)` or `\[ ...\])` , always use braces around the variable. For example write `3^{$n}` instead of `3^$n`. In this example if `$n = 2`, both expressions give what you would expect but if `$n=22`, the first gives `3^{22}` while the second gives `3^22`. TEX interprets the latter as "3 squared times 2", not as "3 to the twenty second power".

13.2 Using `function_cmp`

Here are two places in which is it easy to get caught. Neither are difficult to understand; both have happened to me (only once ... so far).

When using `function_cmp`, the function is defined as a string and you have to be careful about `$variable` substitution. For example, if `$a` is a previously defined variable, say with `$a = -2`, you get an error in the following innocuous looking code:

```
$ans = "-$a*sin($a*x)";
ANS(function_cmp($ans));
```

The error would look something like:

```
Error message:

Tell your professor that there is an error in this problem.
ERROR: at line 1465 of file (eval 97) Can't modify constant item in
predecrement at (eval 109) line 1, near "2*" The calling
package is PG_priv
```

The problem is that the string `$ans` becomes `--2*sin(-2*x)`, and `--` is Perl's decrement operator. The solution is very easy, just put a space after the minus sign:
`$ans = - $a*sin($a*x);`. Perl is smart enough to know `- -2` is 2. Defining
`$ans = - ($a)*sin($a*x);` would be another solution.

There is an analogous problem if you write:

```
$ans = "x+$a**2";
ANS(function_cmp($ans));
```

Again if `$a = -2`, this yields `x+-2**2`, i.e. `x - 4` whereas you probably were expecting `x + 4`. The solution is the write `$ans = x+($a)**2;`.

The second place where one can go astray with `function_cmp` comes from forgetting how it is that two functions are tested for "equality". The functions are not analyzed symbolically; values are tested in each function and compared. The rub comes from the simpler invocations of `function_cmp` (see Sample Problem 2 in the section "Learning to Walk" for more details). That is, the default values are chosen from the interval $(0, 1)$. If the domain of your function does not contain this interval, errors will fly.

Consider the specific example:

```
$ans = sqrt(1- $a x^2);   ## Here $a can be bigger than 1
ANS(function_cmp($ans));
```

This will likely lead to errors. Instead, specify the allowable domain as in

```
$upper_limit = 1/sqrt(abs($a));
ANS(function_cmp($ans,"x",0, $upper_limit));
```

Note in this construct, the independent variable `x` must be specified.

Chapter 14

Answer Evaluators

It is often the case that one asks the students for two answers, and there is no obvious order in which the students should enter them. For example, the general solution to the second order differential equation $y'' - y = 0$ has the form $c_1 f_1(t) + c_2 f_2(t)$. Enter the functions $f_1(t)$ and $f_2(t)$.

I include a portion of such a pg-problem in which it does not matter the order in which students enter their answers. The creator of the problem, must store the answers, test them to see which is which, then then proceed to process them for correctness.

Key to this process is a subroutine called (in this problem) `calleval`, compliments of Zig Fiedorowicz (Ohio State University). It has been written to work with all versions of WeBWorK.

```
DOCUMENT();
loadMacros("PG.pl",
           "PGbasicmacros.pl",
           "PGchoicemacros.pl",
           "PGanswermacros.pl",
           "PGauxiliaryFunctions.pl",
           );

sub calleval {
   my ($ans_evaluator,$student_answer) = @_;
   my $rh_ans_hash = "";
   if  ( ref($ans_evaluator) eq 'AnswerEvaluator' ) { # new style (WW 1.7)
           $rh_ans_hash = $ans_evaluator->evaluate($student_answer);
   } elsif (ref($ans_evaluator) eq  'CODE' )     { #old style (WW 1.6)
           $rh_ans_hash = &$ans_evaluator($student_answer);
   } else {
           warn "There is a problem using the answer evaluator";
   }
   return $rh_ans_hash;
}
```

```
$root1 = non_zero_random(-10,10);
$root2 = $root1 + random(1,5);
## If $root2 is zero, then $root1 < 0
if ($root2 == 0) {$root2 = random(1,10);}
$function1 = "exp($root1*t)";
$function2 = "exp($root2*t)";

## Ok, we are ready to begin the problem...
##
TEXT(beginproblem());


BEGIN_TEXT
$BR
Find the general solution to the homogeneous differential equation
\[
    \frac{d^2y}{dt^2} - \{$root1 + $root2\}\frac{dy}{dt} + \{$root1 * $root2\}y = 0
\]
$BR
The solution has the form
\[y = C_1 f_1(t) + C_2 f_2(t)\]
with
\(f_1(t) = \) \{NAMED_ANS_RULE(first_fcn,25)\}
and \(f_2(t) = \) \{NAMED_ANS_RULE(second_fcn,25)\}
$BR


$PAR
END_TEXT

## Answer Evaluator for value of $function1 at t = 1
($ans_eval1) = num_cmp(exp($root1));

## Grab the first answer entered by student
$firstAnswer = $inputs_ref->{first_fcn};

## Make sure it is defined
$firstAnswer = '' unless defined($firstAnswer);

## Globally substitute (1) for t in $firstAnswer
$firstAnswer =~ s/t/(1)/g;
```

```
## Check first answer against given
$rh_ans_hash = &calleval($ans_eval1,$firstAnswer);


## Check answers accordingly
if ( $rh_ans_hash->{score} == 1 )
{
    NAMED_ANS( first_fcn, fun_cmp($function1, vars=>['t']) );
    NAMED_ANS( second_fcn, fun_cmp($function2, vars=>['t']) );
}
else
{
    NAMED_ANS( first_fcn, fun_cmp($function2, vars=>['t']) );
    NAMED_ANS( second_fcn, fun_cmp($function1, vars=>['t']) );
}

ENDDOCUMENT();
```