WeBWork Authoring

Submitted by:

Justin Gagne
(gagnej3@wit.edu)
Computer Science

Submitted to: Professor Steven Morrow
Department of Applied Mathematics
Wentworth Institute of Technology
July 22, 2016

Table of Contents

1.0 Introduction

This section provides general information regarding WeBWork and the programming languages used for authoring problems. The purpose of this document is presented along with all vocabulary required for a full understanding of the document's content.

1.1 Purpose

This document provides an introduction to authoring homework problems for WebWork using PGML, Perl, and LaTeX. Various objects, functions, and macros used by the WeBWork PG compiler are discussed, as well as the structure and standard conventions to be used when writing a PG homework problem.

1.2 Background

Section 1.2 provides background information on WeBWork and PGML.

1.2.1 WeBWork

WeBWork is an open source homework system provided by and maintained by the Mathematical Association of America (MAA)[1]. WeBWork is used by many schools, primarily throughout the United States, for assigning customized online homework sets to students. Schools that use WeBWork are able to host their own servers for organizing courses and homework sets, storing problems, and keeping track of homework grades. Problems for homework assignments can either be obtained through the MAA's Open Problem Library (OPL) or they can be written and stored on local servers. The OPL contains math problems from a variety courses. Users with administrative access can access the OPL problems and use them within their own homework sets. Administrators also have the ability to write their own problems to be assigned to students.

1.2.2 PGML

PGML, or PG markup language, is a markup language used by the MAA to display problems on WeBWork. Like other markup languages, any content inside a PGML can only be rendered, not executed. Anything that is included in a PGML block is solely to provide a user interface for presenting problems, hints, and solutions. PGML is an updated version of the PG language that was initially used by WeBWork for rendering problems. The markup language itself is available as one of the many dependencies or macros that can be included in a PG file. WeBWork supports backwards compatibility for PG, meaning problems can still be rendered using PG syntax. This document will focus on using PGML for WeBWork authoring as it is a simplified version of PG that provides better readability.

In order to begin authoring problems for WeBWork an author should also become familiar with Perl and LaTeX. A brief introduction to these languages is provided later in this document. In order to account for the lack of robustness that PGML can provide, WeBWork

allows authors to use Perl for handling computations and LaTeX typesetting for displaying relevant formulas and equations. Any computations to be done in Perl must be executed prior to the PGML block they are intended to be used in. Any LaTeX typesetting must be included inside a PGML block.

1.3 Definitions and Acronyms

The following section defines all vocabulary required for an adequate understanding of WeBWork authoring.

1.3.1 Definitions

Section 1.3.1 provides all definitions of terms used in this document. The vocabulary is grouped in the following order:

1. General Programming
    - Defines terms relevant for an understanding of general programming terms and concepts.
2. Object-Oriented Perl
    - Defines any Perl objects or methods used for authoring WeBWork problems.
3. PG Files
    - Defines reserved words, sections, methods, functions, and tags used in a PG file.
4. WeBWork
    - Defines terms relevant to the WeBWork homework system.
5. WeBWork User Types
    - Defines differences of various WeBWork users.

General Programming

Block

- A section of code that is grouped together. The syntax used to group the code is language specific.

Class

- A complex datatype. A class may contain methods, variables, and data structures.

Data Type

- Refers to the type of data that raw data is deemed to be. Also refers to the type of raw data a variable points to. Most languages support basic data types for storing whole number values (integers), real number values (floating point numbers), boolean values (true or false), characters (single character values such as ‘c’), and strings (string data such as “data”).

Index

- A data structure used by a database to point to the location of certain data. Data that is indexed can be accessed at a higher efficiency given proper indexing techniques have been used.

Indexing

- A way of increasing the efficiency of data retrieval when using a database management system. Indexing sorts data based on programmer defined indices to increase the accessibility of certain data.

Inference Type-Checking

- A way that a compiler determines data types of variables. Variables are not assigned a data type until they have a value assigned to the identifier.

Member Variable

- A variable that is declared or defined within a class.

Method

- A block of reusable code that call be called to execute a specific action.

Object

- A variable of a predefined class type.

User Interface

- Content that is visible to a user and potentially available for user interaction. In this context, the user interface refers to a homework problem that is rendered and displayed to a user on WeBWork.

Tag

- A reserved word in a markup language that is used for describing how content is to be rendered.

Variable

- A programmer defined keyword used for accessing and referencing data during the runtime of a program.

Widget

- An object that is linked to a specific user interface layout.

Object-Oriented Perl

Array

- One of the three basic data types of Perl. A basic array can point to any number of scalar values. An array variable is declared by preceding the identified with an '@' symbol [7].

Context

- A Perl object that defines the type of numbers used for a PG problem. Every PG problem is given a default Context unless otherwise specified by the author. Also referred to as the "Context" of a problem or the Context object.

Formula Object

- A Perl object that converts a string containing variables and numbers to a computable mathematical formula.

Hash

- One of the three basic data types of Perl. A hash variable provides an unordered set of key-value groupings. Hash variables are declared by preceding an identifier with the '%' symbol.

RadioButton

- A Perl widget that provides a user interface for multiple answer questions.

Scalar

- One of the three basic data types supported by Perl. A variable of type scalar can be assigned to either a numerical value or a string value. To define a scalar type variable, the identifier should be preceded by a dollar sign ($) before assigning the a value to the variable[7].

Tolerance

- A member variable of the Context class that is used to define how much the user's answer can vary from the actual answer of the problem. This value can be defined within the Context of a PG problem to account for rounding discrepancies.

tolType

- A member variable of the Context class that is used to define the interval of the tolerance.

PG Files

BEGIN_PGML

- The tag used to mark the beginning of a PGML block that will be rendered and visible to the user.

BEGIN_PGML_HINT

- The tag used to mark the beginning of a PGML hint block that will be accessible to the user after a certain number of incorrect attempts.

BEGIN_PGML_SOLUTION

- The tag used to mark the beginning of a PGML solution block that will be accessible after the due date of a homework assignment.

DESCRIPTION

- The tag used to mark the beginning of a PG description block.

DOCUMENT()

- A function call that must be included in a PG file to indicate the beginning of the executable section of the file.

ENDDESCRIPTION

- The tag used to mark the end of a PG description block.

ENDDOCUMENT()

- A function call that must be included in a PG file to indicate the end of the executable section of a PG file.

END_PGML

- The tag used to mark the conclusion of a PGML block.

END_PGML_HINT

- The tag used to mark the conclusion of a PGML hint block.

END_PGML_SOLUTION

- The tag used to mark the conclusion of a PGML solution block.

loadMacros()

- A function call to load dependencies to the PG compiler when a problem is being compiled.

Macros

- File dependencies that can be referenced in a PG file. Macros allow authors to access various libraries that have been written for WeBWork. Available libraries range from Perl-PG widgets to computational libraries for answer checking.

PG File

- A file with the extension ".pg". PG files may contain Perl code, Python code, PG code, and/or PGML code. These are the files that are rendered by WeBWork to display problems on the user interface.

PG Header

- The beginning of a PG file. The header is a section of comments used by the WeBWork OPL for indexing and searching for homework problems.

Renderable Section

- Content that is contained between either the BEGIN_PGML and END_PGML tags, BEGIN_PGML_HINT and END_PGML_HINT, BEGIN_PGML_SOLUTION and END_PGML_SOLUTION tags. These are the portions of a PG file that will be displayed to the user.

WeBWork

Local Directory

- A directory accessible by a specific college or university. Problems can be saved on a school's local server and used in courses belonging to the school when creating and assigning homework sets.

Local Server

- A server hosted by a school that can be used to store problems written by or belonging to the school. Users with administrative privileges can store problems or relevant data in the local server. Courses under the school's account are granted access to the files stored on the local server and can use the local files in homework sets.

WeBWork

- An open source homework system provided by the MAA.

WeBWork User Types

Administrator

- A user who has access to alter course information and file directories on a WeBWork account. Users with administrative privileges are typically WeBWork authors or professors at a college or university.

Author

- A user who writes problems to be used on WeBWork. An author must have administrative privileges to the local server to upload problems.

Student

- A user who is assigned homework problems on WeBWork but does not have administrative privileges.

User

- Someone who views or uses WeBWork.

1.3.2 Acronyms

MAA

- Mathematical Association of America. The association that provides and maintains WeBWork

OPL

- Open Problem Library. A database hosted by the MAA that stores various homework problems in mathematics. WeBWork provides access to the OPL for administrators to add problems to a homework set assigned to students. As of 2015, the OPL contained approximately 25,000 problems available to WeBWork users [2].

PGML

- PG Markup Language. A markup language used for displaying homework problems on WeBWork.

UI

- User Interface. Content that is displayed to a user.

1.4 References

[1] Welcome to WeBWorK. (n.d.). Retrieved June 21, 2016, from http://webwork.maa.org/

[2] Open Problem Library. (2015, December 13). Retrieved July 09, 2016, from http://webwork.maa.org/wiki/Open_Problem_Library#.V4FHcJMrJTY

[3] Using WeBWork. (n.d.). Retrieved July 09, 2016, from http://webwork.maa.org/moodle/mod/data/view.php?d=1

[4] Introduction to Contexts. (n.d.). Retrieved July 09, 2016, from http://webwork.maa.org/wiki/Introduction_to_Contexts#.V1nxn-YrJTY

[5] Current DBchapter and DBsection Tags. (n.d.). Retrieved July 16, 2016, from
http://hobbes.la.asu.edu/Holt/chaps-and-secs.html
List of subject, chapter, and section names for WeBWork
[6] Current Keywords. WeBWork. Retrieved July 16, 2016, from
http://hobbes.la.asu.edu/Holt/keywords.html Keywords used for WeBWork indexing.
[7] Perl Data Types. (n.d.). Retrieved July 21, 2016, from
http://www.tutorialspoint.com/perl/perl_data_types.htm

2.0 PG Template

Section 2 discusses the various sections of a PG file and the uses of each section.

2.1 PG Header

This section should appear as the first section of a PG file. The header is composed of comments containing information used by the OPL for indexing and searching for various problems. The header should include citation details such as a textbook name, chapter, section, problem number, names of the textbook authors, and the name of the PG file author. An example header is provided in Figure 1.

```
## DESCRIPTION
## An example template for authoring WeBWork problems using Perl and PGML.
## ENDDESCRIPTION

## Author('Justin Gagne')
## DBcourse('The course name in OPL or local server')
## DBchapter('A chapter name of a course')
## DBsection('A section name of a chapter')
## KEYWORDS('Any', 'keyword', 'relevant', 'to', 'the', 'problem')
## TitleText1('The title of the textbook')
## EditionText1('1')
## Section1('5-2')
## Problem1('43')
## AuthorText1('Contributing author of TitleText1')
## AuthorText1('Contributing author of TitleText1')
## Institution('Wentworth Institute of Technology')
```

Figure 1: PG Header

2.1.1 Description and Author Tags

The header of a PG file should contain the description of the problem enclosed within the PG description tags. The remainder of the header is used for referencing the source in which the problem was obtained from. The Author tag refers to the person who wrote the PG file.

2.1.2 DB Course, Chapter, and Section Tags

DBcourse is the most general tag used for indexing. This refers to the name of the course in which the problem belongs to. Example course names include tags such as ‘Algebra’ and

‘Calculus - single variable’. The DBchapter tag refers to a general subject of a specified course. If the course of a problem is ‘Calculus - single variable’, a valid chapter name is ‘Applications of differentiation’. The most specific subject tag is DBsection. This tag refers to a subcategory of a DBchapter tag. Using the previous chapter example of ‘Applications of differentiation’, an example of a valid section tag is ‘Rates of change - general’[5]. A complete list of valid tags for indexing authored WeBWork problems can be found in the link included in reference [5]. Tags for WeBWork problems should be obtained from the provided link to ensure proper problem indexing, especially for problems submitted to the OPL.

2.1.3 Keywords

The KEYWORDS tag is used for listing indexing terms relevant to the problem itself. All keywords must be single words contained in single quotation marks. The KEYWORDS tag can contain any number of comma separated keyword values. The contents of this tag are used to help other users search for a specific problem when searching the OPL. A list of all current WeBWork keywords can be found by following the link provided in reference [6].

2.1.4 Header Citations

All header citations should follow the TagNameN convention. The tag name refers to the type of information being cited and *N* refers a number from 1-*N*, of *N*-sources being cited by the problem’s author. Using this convention, the TitleText1 tag is used to cite the name of the first textbook, TitleText2 would refer to a second text book, and so forth. Knowing this, each of the remaining tags corresponding to information obtained from the *N*-th textbook should end with the *N*-th number which the information corresponds to. To cite the remaining information, EditionTextN is used to cite the edition of a textbook, SectionN is used to cite the chapter-section combination of the textbook, ProblemN is used to cite the number of the problem, and AuthorTextN is used to cite the names of contributing authors of the textbook. At the end of citing all textbooks used, the name of the college or university the PG file was written for should be included in the Institution tag.

2.2 The Executable Section

The executable section refers to the portion of a PG file that is to be compiled. This includes all content contained between between the DOCUMENT() and ENDDOCUMENT() function calls. All code that is required for the problem should be included between these two function calls. Any PG file that does not contain these function calls will yield errors when WeBWork attempts to compile the problem. Example syntax of declaring the executable section of a PG file is provided below in Figure 2.

```
DOCUMENT();        # This should be the first executable line in the problem.

    ##All code should be included between these function calls, excluding the PG header

ENDDOCUMENT();     # This should be the last executable line in the problem.
```

Figure 2: Executable Section

2.3 Loading Macros

Authors can choose to include any number of dependencies in a PG file. Dependencies in WeBWork are referred to as macros. Marcos are used to add flexibility to the UI and backend of a problem. Any macros necessary for proper compilation of a problem should be listed directly after the DOCUMENT() function call in the executable section. To tell the PG compiler that macros must be loaded, the loadMacros() function must be called. This function can take any number of comma-separated string parameters referring to a ".pl" file in the OPL. As of July 2016, WeBWork has 914 available macros that can be used by authors [3]. Since PGML itself is a dependency it is crucial to include the "PGML.pl" file in the loadMacros() function. An example of how to use the loadMacros() function is shown below in Figure 3.

```
DOCUMENT();        # This should be the first executable line in the problem.

#Add any required plugins here. All plugins must be comma separated except for the last one.
loadMacros(
"PGstandard.pl",
"PGbasicmacros.pl",
"PGchoicemacros.pl",
"PGanswermacros.pl",
"PGauxiliaryFunctions.pl"
);

loadMacros("PGML.pl");
```

Figure 3: Loading Macros

The first group of macros should always be included as they provide the most basic plugins for PG rendering. The second group of macros is used to include the PGML macro to allow for simplified PG file authoring.

2.4 Defining Context

Once all required macros are listed, the context of the problem should be set. Context defines the values that the PG compiler will accept such as variable names, global constants, and number types. Context is also used for defining the tolerance to be used for answer checking. Each problem that is compiled is given a default Context of type "Numeric", a context variable

of ‘x’, and a tolerance value of 0 unless otherwise specified. The most common contexts used include “Numeric, Complex, Vector, Matrix, and Interval” [4]. The context of a problem should be known and defined by the author to avoid compilation errors. Figure 4 (below) provides an example of adding variables to the Context of the problem.

```
#Add a new variable 't' to the context of the problem. Any letters can be added
#to the context and used as variables. In this case, the Context tells the PG
#compiler to recognize 't' as a variable. The variable 't' represents an arbitrary
#'Real' number.
Context()->variables->add(t => 'Real');

#Variables can be removed from the Context
Context()->variables->remove("t");

#Multiple variables can also be added at the same time if separated by commas.
Context()->variables->add(t=>'Real', y=> 'Real');
```

Figure 4: Context Variables

The figure above gets access to the Context object by calling Context(). The sample code accesses the member array ‘variables’ and calls the add method to add the variable ‘t’ as a Real-type variable. Multiple variables can be added at the same time or removed from the Context. Defining Context variables is a good idea if the problem is not well represented by the default ‘x’ variable. For example, an author may want to declare a context variable of ‘t’ if the independent variable represents time. Context can also be altered to remove, add, or redefine constant values of the problem. Figure 5 (below) removes the default ‘pi’ constant and then adds it back into the Context using a shortened value for ‘pi’.

```
#'pi' is a default constant in Context. Remove 'pi' and give it a new value:
Context()->constants->remove("pi");
#Add a new constant 'pi' to the context of the problem
Context()->constants->add(pi=3.14159);
```

Figure 5: Adding Context Constants

As shown in Figure 5, constants can be added or removed from the Context, just like variables. Multiple constants can also be added by separating the constants with commas in the add method. The last notable use of the Context object is the tolerance. Tolerance is used when answer checking (Section 2.8) to allow the compiler to accept user submitted answers within a defined range. Tolerance should be defined for problems that have answers resulting in a non-integer value. This allows users to submit answers that are close to the calculated answer but

may have slightly different values due to rounding. An example of setting the tolerance of a problem is given below in Figure 6.

```
#Allow for tolerance in the solution. This allows students to enter values within
#the tolerance range
Context()->flags->set(
  tolerance => .0009,
  tolType => "absolute",
);
```

Figure 6: Context Tolerance

To set the tolerance of a problem, the Context 'flags' array must be accessed and the set method must be called to set values for tolerance and tolType. The tolerance variable is a 'Real' value that defines how much the user's answer can vary from the compiler computed answer. The tolType defines how the variance can occur. By setting tolType to "absolute", this means that the answer will be deemed correct if the submitted answer is within the interval (-tolerance, +tolerance) of the computed answer.

2.5 Problem Header

The problem header marks the beginning of the actual homework problem. This section is a line of text that is displayed to the user to provide a brief description of the problem. Information typically includes the problem's point value and title. The point value is the first information displayed. It should be contained in parentheses and followed by the title of the problem in bold. An example problem header is displayed below in Figure 7.

```
#Display a title for the problem
TEXT( "(1 pt) " , $BBOLD, "Introduction to Formulas", $EBOLD, $BR, $BR);
```

Figure 7: Problem Header

The problem header is set by calling the PG TEXT function. This function can contain any number of parameters. In the provided example, the header will displayed as:

(1 pt) **Introduction to Formulas**

The title is then followed by two line breaks. The problem header is usually programmed using PG syntax rather than PGML to allow authors to easily distinguish where the problem begins. Since the header uses PG syntax, the $BBOLD (begin bold) value is used to set all following text

in bold font until the $EBOLD (end bold) parameter is reached. Each $BR parameter tells the compiler to print a line break to be displayed in the rendered problem.

2.6 Declaring Variables

All Perl variables relevant to the problem should be defined after the problem header. Perl is an *inference* based programming language meaning a variable's data type does not need to be declared. Instead, the compiler will assign a data type to a variable once it is assigned to some form of data. In Perl, the most basic data type is scalar. Scalar data is either stored as integer, a real number, or as a string. No errors will be produced when assigning a variable pointing to string data to a variable pointing to numerical data. Instead, the variable being assigned data will take on the new data type of the data being assigned. Dynamic casting from string values to numerical types should be avoided to reduce the likelihood of computation errors.

Perl also provides support for array variables and hashes. Array variables can have any number of indices that point to a scalar data type and can be declared by beginning the identifier with an '@' symbol. The last basic data type, hashes, can be declared by preceding the identifier with a "%" symbol. Hashes are "unordered sets of key/value pairs" that can be accessed using their key value [7]. Figure 8 (below), provides an example of declaring variables of different basic types supported by Perl. All variable declarations should end with a semicolon.

```
#Scalar Data Types
$real = 4.30;                   #Scalar variable with data stored as a double
$integer = 1;                   #Scalar variable with data stored as an integer
$string = "Some string data";   #Scalar variable with data stored as a string

#Arrays
@numberArray = (1,2,3.5);               #Array of numerical values
@stringArray = ("String1", "String2");  #Array of string values
@mixedArray = ("string", 1, 2.3);       #Array of mixed data

#Hashes
%wordToNum = ('One', 1, 'Two', 2);   #Hash to map 1-->'One' and 2-->'Two'
```

Figure 8: Declaring Variables

Since scalar values are the most basic data type, a scalar value can be referenced directly by its variable name. Accessing specific data from an array or hash is slightly more complicated. Values can be obtained from an array by referring to the scalar value at a certain index. To get data from a hash, the scalar value of a key must be referenced. An example of accessing values from arrays and hashes is provided in Figure 9 (next page).

```
#Getting data from arrays
$num1 = $numberArray[0];
$num2 = $numberArray[1];
$num3 = $numberArray[2];

#Getting data from hashes
$one = $wordToNum{-One};  #Get the value for the key 'One'
$two = $wordToNum{-Two};  #Get the value for the key 'Two'
```

Figure 9: Accessing Array and Hash Data

To access data from arrays, the array identifier must be referred to as a scalar variable and the desired index number must be placed in brackets. Array indices are accessible from 0 to n-1, where n is the number of elements in the array. To access data from hashes, the hash identifier must be casted to a scalar and the value of the desired key should be placed in curly brackets beginning with a hyphen.

2.7 Declaring Formulas

For certain problems it may make sense to include various Formula objects to increase the flexibility of the problem or to simplify the computation of the solution. Formula objects are instantiated very similarly to scalar variables. A variable must be declared beginning with a '$' character and then assigned to a Formula constructor method to become a Formula object. Consider the equations "$f(x) = 3x^2 + 4x + 5$" and "$g(t) = 4t^3 - 10$". These equations will be used as examples in the next few sections. An example of instantiating Formula objects using the given equations is provided in Figure 10 below.

```
#Instantiate a formula object using the formula containing in the quotations
$f = Formula("3x^2+4x+5")   #f(x) = 3x^2 + 4x + 5
$g = Formula("4t^3-10");    #g(t) = 4t^3 - 10
```

Figure 10: Using Formula Objects

Once a Formula object has been instantiated it can be referenced like any other scalar value. Note that any variables used to create a Formula object must be added to the Context before hand. If any undeclared variables are used, the compilation of the problem will yield errors. Once a formula is stored in an object, it can be displayed in a block of PGML the same way as a scalar variable (Section 2.9). Formulas can also be used to quickly substitute a numerical value for a variable in the equation. An example of substituting values into the previously discussed equations is presented in Figure 11 (next page).

```
#Find the initial condition of both of the formulas f and g.
$f_initial = $f->substitute(x,0);   #f(0) = 3(0)^2 + 4(0) + 5
$g_initial = $g->substitute(t,0);   #g(0) = 4(0)^3 - 10
```

Figure 11: Substituting into Formulas

To avoid losing the initial formula, a new object should be instantiated and assigned to the equation containing the substituted values. To substitute for a variable in a formula, the substitute method is called. The first parameter is the name of the variable to substitute for and the second parameter is the number that is to be substituted into the equation. When 0 is substituted into both of the formulas discussed above, the result is as follows.

$f_initial = "$3(0)^2 + 4(0) + 5$"

$g_initial = "$4(0)^3 - 10$"

The two new formulas contain no variables. Instead, the variables have been substituted with the constant stated in the substitute method. The equations have not however been simplified or solved. Each of the new objects, $f_initial and $g_initial, point to an equation that has 0 in all locations that previously contained a variable.

2.8 Computing Solutions

Once all relevant formulas and variables have been defined, the solution to the problem should be computed. To compute a solution the Compute() function should be called. Compute can take either a Formula or a hard-coded equation of Perl variables as parameters. If the parameter used is a Formula, a numerical value must have been substituted into the Formula prior to passing it as a parameter to the Compute function (see Section 2.7 for how to substitute into Formulas). If the parameter used is a hard-coded equation using Perl variables, all variables must be defined. If the Compute function accepts the given parameters, it will return a solution to the equation based on the Context of the problem. An example of computing the solution of a function object is shown in Figure 12 (next page).

```
#Instantiate a formula object using the formula containing in the quotations
$f = Formula("3x^2+4x+5");   #f(x) = 3x^2 + 4x + 5
$g = Formula("4t^3-10");    #g(t) = 4t^3 - 10

#Find the initial condition of both of the formulas f and g.
$f_initial = $f->substitute(x,0);   #f(0) = 3(0)^2 + 4(0) + 5
$g_initial = $g->substitute(t,0);   #g(0) = 4(0)^3 - 10

#Compute the solution to formula with substituted values
$ans_f_initial = Compute($f_initial); #f(0) = 5
$ans_g_initial = Compute($g_initial); #g(0) = -10
```

Figure 12: Using the Compute Function

Once the Compute function is executed, it stores the value of the result in the variable it is assigned to. The result of each step is included in the comments of the code-snippet in Figure 12. The same result can be achieved by hardcoding an equation in Perl. This approach tends to be the less elegant since mathematics in Perl can lead to complicated looking code. An example of a hard-coded computation is presented in Figure 13.

```
#Compute the solution for the initial condition using a hardcoded equation
$x = 0;                                  #Initial condition of x
$t = 0;                                  #Initial condition of t
$f_initial_2 = 3*(($x)**2) + 4*$x + 5;  #f(0) = 3(0)^2 + 4(0) + 5 = 5
$g_initial_2 = 4*(($t)**3) - 10;        #g(0) = 4(0)^3 - 10 = -10
```

Figure 13: Computing Solutions

Figure 13 presents hard-coded solution to getting the initial values of the two functions f(x) and g(t). Although it requires less lines of code to solve for the initial condition in this case, this approach should be avoided and Formulas should be used whenever possible.

2.9 Displaying the Problem

Once all the background code has been written, the actual problem that is seen by the user can be written. In this part the renderable section is discussed. The renderable section defines text between the BEGIN_PGML and END_PGML tags. In this part of a PG file, it is possible to include text, LaTeX formulas, Perl variables, and Perl objects. Figure 14 (next page) shows the shell of a renderable section.

```
#Marks the beginning of a section of text/code that is to be displayed to the user when rendered.
BEGIN_PGML

    [%Any content to be displayed to the user goes here%]

END_PGML
#Marks the end of a section of text/code that is to be displayed to the user when rendered.
```

Figure 14: Renderable PGML Block

All content inside a PGML block will be visible to the user except comments (anything contained in [%Text goes here%]). There is no specific format or notation required for including plain text in PGML. All text in this block will be displayed exactly as the author has entered it. For an example problem, the user will be prompted to solve for the initial conditions of the previous example functions "$f(x) = 3x^2 + 4x + 5$" and "$g(t) = 4t^3 - 10$". Since this tutorial has previously defined Formula objects for both of the functions, those can be included in the PGML block. Recall that f(x) and g(t) were defined as $f and $g in Perl respectively. To include a Perl variable or object in PGML, wrap the identifier with using square brackets. Refer to the code snippet in Figure 15 for how to include variables in PGML and Figure 16 for how the PGML block appears to the user. Note that: [% %] is used for including comments inside the renderable section.

```
#A section of text/code that is to be displayed to the user when rendered.
BEGIN_PGML

  Given:
    f(x) = [$f]     [%Show the formula for $f%]
    g(t) = [$g]     [%Show the formula for $g%]

  Find the initial conditions of f(x) and g(t).

END_PGML
```

Figure 15: PGML Example Code

(1 pt) **Introduction to Formulas**

Given:

f(x) = 3*x^2+4*x+5
g(t) = 4*t^3-10

Find the initial conditions of f(x) and g(t).

Figure 16: PGML Example Rendered

Notice that when the equation is rendered in Figure 16, it is difficult to read. This is because all formulas and equations for a problem should be included in a LaTeX typesetting box. To include LaTeX inside a PGML block, enclose text the desired text using [` `]. Anything between the backticks will be displayed using LaTeX typesetting when it is rendered. To include more complex formulas, the typesetting can be hard-coded into the block. To see this example updated using LaTeX see Figure 17 for the code and Figure 18 for the rendered content.

```
#A section of text/code that is to be displayed to the user when rendered.
BEGIN_PGML

  Given:

    [`f(x) = [$f]`]      [%Show the formula for $f%]
    [`g(t) = [$g]`]      [%Show the formula for $g%]

  Find the initial conditions of [`f(x)`] and [`g(t)`].

END_PGML
```

Figure 17: PGML Example Code with LaTeX

(1 pt) **Introduction to Formulas**

Given:

$$f(x) = 3x^2 + 4x + 5$$
$$g(t) = 4t^3 - 10$$

Find the initial conditions of $f(x)$ and $g(t)$.

Figure 18: PGML Example Display with LaTeX

By including the LaTeX equation, the readability of the rendered problem is greatly increased. LaTeX should only be used for displaying the equations, not the textual description of the problem. This also shows one of the conveniences of the Formula object, when it is included in a LaTeX block, it is automatically rendered. If the equation was displayed using solely Perl variables, the author would have to manually include the LaTeX typesetting in the LaTeX block.

Now that the problem is more readable, there is one more task that the PGML block must accomplish. There must be somewhere for the user to enter the solution. To display a text-input box for the user to interact with, include any number of underscores enclosed by square braces in the PGML block. The number of underscores included corresponds to the length of the entry field displayed to the user. There should be a solution field for each part of the problem. To check the correctness of an answer entered in the text field, include the following code:

[_____]{$answer} [%the answer corresponding to a part of the question%]

By adding this line, there will be a text field that corresponds to the $answer variable. For a problem with multiple answers, there should also be a variable for each part of the solution. Figure 19 shows the final code for the PGML block with answer checking and Figure 20 shows how it looks when rendered on WeBWork (below).

```
#A section of text/code that is to be displayed to the user when rendered.
BEGIN_PGML

  Given:

    [`f(x) = [$f]`]      [%Show the formula for $f%]
    [`g(t) = [$g]`]      [%Show the formula for $g%]

  Find the initial conditions of [`f(x)`] and [`g(t)`].

  [`f(0) = `] [_____]{$ans_f_initial}

  [`g(0) = `] [_____]{$ans_g_initial}

END_PGML
```

Figure 19: Complete PGML Example Problem Code

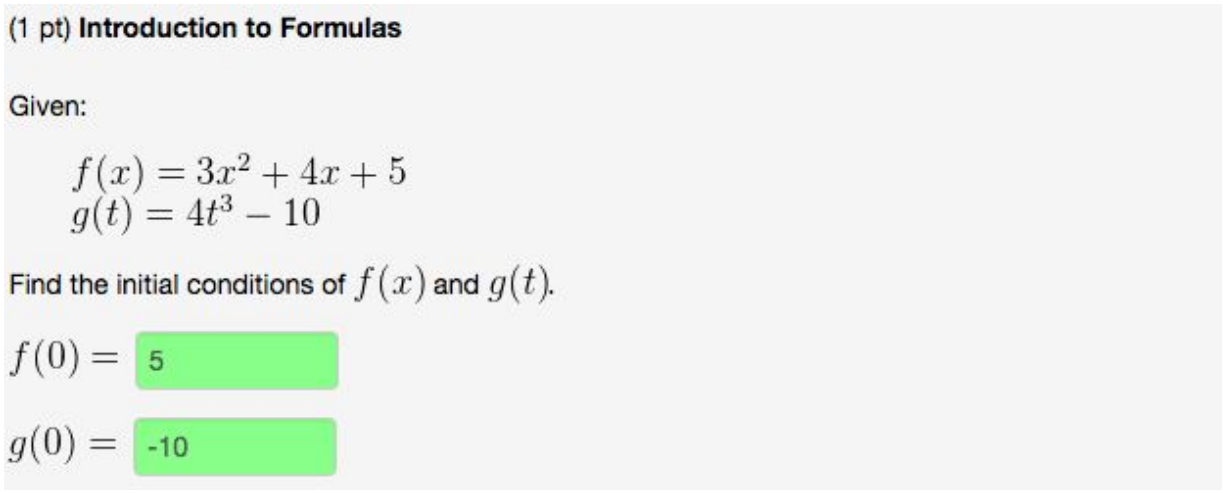

Figure 20: Complete PGML Example Problem Rendered

Figure 19 shows the the complete example problem. Each of the text fields included in the PGML block have a unique answer corresponding to the solution of part of the problem. Figure 20 shows how the problem is displayed when it is compiled for WeBWork. The options for the user to submit the solution are included on the WeBWork website so the author does not need to include the submit buttons. The correctness of the PG file can be tested by inputting the

answers to the problem and clicking the either “Check Answers” or “Submit Answers” button on WeBWork. Since the text fields are highlighted green, it can be confirmed that the problem works and can be used on homework assignments for students.

2.10 Displaying Hints

It is possible to display hints to students who are having difficulty with correctly answering the problem. The hint PG code for the hints block should be placed after the actual problem in the PG file. Hints are not displayed to the student until an administrator defined attempt count is reached. To include hints to the problem, a new renderable block is to be programmed using the BEGIN_PGML_HINT and END_PGML_HINT tags. The hint block works exactly the same was as a plain PGML block, except it is not visible to students until the problem attempt limit is reached. Figure 21 shows how to include a hint in a PG file, Figure 22 shows the hint link that is added, and Figure 23 shows how the hint looks when rendered to the user.

```
#Mark the start of a renderable PGML hint block.
BEGIN_PGML_HINT

  An initial condition of an equation can be found by evaluating an equation at 0.

  For each [`x`] in [`f(x) = [$f]`], substitute [`x = 0`]. [%Show the formula in LaTeX%]

  For each [`t`] in [`g(t) = [$g]`], substitute [`t = 0`]. [%Show the formula in LaTeX%]

END_PGML_HINT
#Mark the end of a renderable PGML hint block.
```

Figure 21: Sample Code for PGML Hint Block

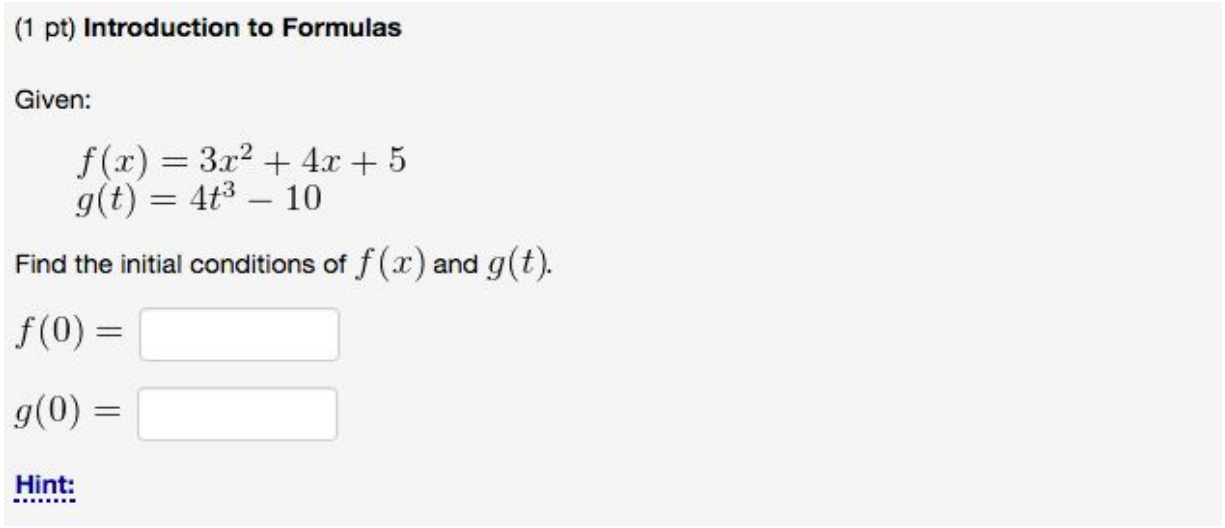

Figure 22: Hint Link

(Instructor hint preview: show the student hint after 1 attempts. The current number of attempts is 0.)
An initial condition of an equation can be found by evaluating an equation at 0.

For each x in $f(x) = 3x^2 + 4x + 5$, substitute $x = 0$.

For each t in $g(t) = 4t^3 - 10$, substitute $t = 0$.

Figure 23: Rendered PGML Hint

Once the PGML hint block has been added to the PG file (Figure 21), a link for the hint can be seen at the bottom of the problem (Figure 22). When the user clicks on the link, a drop down text block appears on WeBWork. The drop down is the rendered content from the PGML hint block which can be seen on the website (Figure 23).

2.11 Displaying Solutions

The final section of a PG file should be the PGML solution. The solution, by default, is not available to students until after the deadline of a homework set. This attribute can be updated by an administrator. To add a solution to a PG file, the BEGIN_PGML_SOLUTION and END_PGML_SOLUTION tags are used. All content for the the solution is to be placed within these tags and follows the same syntax as the plain PGML blocks. A solution is not required in a PG file but it can be helpful for students who do not understand the material. Using the same problem from the previous examples, PGML solution sample code is provided in Figure 24, the added solution link is shown in Figure 25, and the rendered solution is can be found in Figure 26.

```
#Mark the start of a renderable PGML solution block.
BEGIN_PGML_SOLUTION
Solve for the initial conditions of the given equations:

  1). [`f(x) = [$f]`]                [%Show the initial formula in LaTeX%]

      [`f(0) = [$f_initial]`]        [%Show the formula with substitutions in LaTeX%]

      [`f(0) = [$ans_f_initial]`]    [%Show the answer in LaTeX%]

  2). [`g(t) = [$g]`]                [%Show the initial formula in LaTeX%]

      [`g(0) = [$g_initial]`]        [%Show the formula with substitutions in LaTeX%]

      [`g(0) = [$ans_g_initial]`]    [%Show the answer in LaTeX%]

END_PGML_SOLUTION
#Mark the end of a renderable PGML solution block.
```

Figure 24: Sample PGML Solution Code

Figure 25: Sample PGML Problem with Solution Link

Figure 26: Rendered PGML Solution

Once the PGML solution block has been added to the PG file (Figure 24), a link for the solution can be seen at the bottom of the problem (Figure 25). When the user clicks on the link, a drop down text block appears on WeBWork. The drop down is the rendered content from the PGML solution block which can be seen on the website (Figure 26). Solutions, when included, should have a step-by-step procedure to completing the problem.