

THE ADAPT WEBWORK HANDBOOK



The ADAPT WeBWork Handbook
Roosevelt University

The ADAPT WeBWork Handbook

Neil R. Voss

Roosevelt University

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 03/03/2026

TABLE OF CONTENTS

AI Disclaimer

Licensing

1: Introduction

- 1.1: What is WeBWork?
- 1.2: Why use WeBWork?
- 1.3: Key Features of WeBWork
- 1.4: Comparing WeBWork to other formats
- 1.5: Common Terms and Names
- 1.6: Quickstart - Edit Your First Problem

2: Problem Generation (PG)

- 2.1: Introduction to PG Language
- 2.2: OPL Header and Metadata
- 2.3: PG Problem Files with a .pg file extension
- 2.4: Sections within a PG question
- 2.5: Common PG Macros
- 2.6: Legacy PG and deprecated patterns
- 2.7: Future PG version features

3: PGML (PG Markup Language)

- 3.1: Introduction to PGML
- 3.2: Answer blanks and answers
- 3.3: Lists and workflows
- 3.4: Tables and Structured Data
- 3.5: Layout controls
- 3.6: Command substitution and inserts

4: Breaking Down the Components of a WeBWork Problem

- 4.1: Full file
- 4.2: OPL Header
- 4.3: Preamble
- 4.4: Setup
- 4.5: Statement
- 4.6: Solution
- 4.7: Putting It All Together

5: Different Question Types in WeBWork

- 5.1: Question Types Overview
- 5.2: Multiple Choice
- 5.3: Multiple Answer
- 5.4: Matching
- 5.5: Numerical Entry

- [5.6: Fill in the Blank](#)
- [5.7: Multipart Questions](#)
- [5.8: Ordered List](#)
- [5.9: Workflow and QA](#)

6: Advanced PGML Techniques

- [6.1: Text Coloring and Emphasis](#)
- [6.2: Making Tables with niceTables](#)
- [6.3: Making Graphs](#)
- [6.4: Advanced Randomization Techniques](#)
- [6.5: Randomized Matching Problems](#)
- [6.6: Randomized MC True and False Statements](#)
- [6.7: Rendering Chemical Structures with RDKit](#)
- [6.8: Displaying Chemical Formulas](#)

7: Testing and Debugging

- [7.1: Simple Syntax Checking \(Linting\) in WeBWork](#)
- [7.2: Setting Up the PG Renderer](#)
- [7.3: Scripting and Automation of the PG Renderer](#)
- [7.4: Common Mistakes and How to Fix Them](#)
- [7.5: Testing Randomization and Edge Cases](#)
- [7.6: QA Checklist Before Publishing](#)

8: Using AI Agents to Write WeBWork

- [8.1: What are AI Agents?](#)
- [8.2: Knowledge Documents for AI Agents](#)
- [8.3: Connecting AI Agents to the PG Renderer](#)
- [8.4: Advance AI Agents Advice](#)

9: Appendices

- [9.1: Minimal Templates](#)
- [9.2: Troubleshooting Checklist](#)
- [9.3: Macro Demonstrations](#)

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

[Detailed Licensing](#)

AI Disclaimer

This handbook was generated using multiple AI tools from my notes on using WeBWorK in February 2026. Chapter topics were chosen and reviewed by Dr. Neil R. Voss. Code snippets were extracted and tested using a PG renderer to confirm they run, but you should still review and test code before using it in your own course.

This edition is being released before a full line by line review because search engines have already indexed earlier workbench drafts, and it is better to publish a stable version now. The text may contain errors, which will be corrected as they are discovered.

Licensing

Licensing and Attribution

This book, *Writing Automated Questions using WeBWorK in LibreTexts' ADAPT*, is licensed under a **Creative Commons Attribution 4.0 International License (CC BY 4.0)**.

You are free to:

- **Share:** copy and redistribute the material in any medium or format.
- **Adapt:** remix, transform, and build upon the material for any purpose, including commercial use.

Under the following terms:

- **Attribution:** you must give appropriate credit, provide a link to the license, and indicate if changes were made. Attribution should not suggest that the author or LibreTexts endorses you or your use.

There are no additional restrictions. You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

The full license text is available at: <https://creativecommons.org/licenses/by/4.0/>

Third-Party Content

Some examples, code patterns, or excerpts may be adapted from the OpenWeBWorK documentation or other publicly available resources. Those materials remain under their original licenses and are cited where appropriate.

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: Introduction

There are plenty of guides for writing mathematics assessment questions in WeBWorK. This guide takes a different approach and is written for people in the sciences and other descriptive fields, especially in the life sciences.

This book assumes you want questions that read like real lab and lecture tasks, grade reliably, and are easy to reuse across semesters.

This book assumes you are a domain expert, not a software developer. The patterns shown here are designed so that you can reason about biology first and let the templates handle the mechanics.

This guide is based on the PG 2.17 subset used in ADAPT and webwork-pg-renderer, so some macros from full WeBWorK installs are not available. Use the macro allowlist reference in [Sections within a PG question](#) before you try to load new interaction macros.

On this page

On this page: section and what it covers.

Section	What it covers
What this guide emphasizes	The PGML-first mindset, science-first tone, and subset constraints.
PG vs PGML (a practical split)	A concrete rule of thumb for what goes where.
Where to start today	A task-to-chapter map for quick entry.
Apply it today	Two actions you can take immediately.

What this guide emphasizes

- PGML-first authoring: write the student-facing prompt in PGML and keep regular PG as short setup code for values and answer checking.
- Life-science style prompts: short narratives, concentrations and dilution workflows, structured data, and interpretation questions that match how biology is taught and practiced.
- Practical reuse: stable patterns you can clone and copy-edit to build question banks quickly without breaking grading.
- Subset-aware authoring: the ADAPT and webwork-pg-renderer PG 2.17 subset lacks some macros, so check the allowlist in [Sections within a PG question](#) before adding new interaction types.

PG vs PGML (a practical split)

PG vs PGML (a practical split): layer, use it for, and avoid using it for.

Layer	Use it for	Avoid using it for
PG (setup)	Macros, context, variables, answer checkers.	Long prompt text, hand-built formatting.
PGML (prompt)	Student-facing sentences, math, tables, blanks.	Hidden logic and heavy computation.

Where to start today

This guide is organized as short pages you can read out of order. Pick the path that matches what you are doing today.

Where to start today: if you are doing this today, start here, and what you get.

If you are doing this today	Start here	What you get
Make one working problem fast	Quickstart: Edit Your First Problem	A safe copy-edit loop and a minimal working skeleton.
Set up a new file correctly	Problem Generation (PG)	Minimal scaffolding that stays readable.
Write or edit the prompt text	PGML (PG Markup Language)	PGML patterns for blanks, lists, tables.
Understand the anatomy of a full problem	Breaking Down the Components	OPL header plus four sections in one worked example.
Pick an interaction type	Different Question Types	A catalog of common question types.
Build biology-first patterns	Advanced PGML Techniques	Advanced PGML techniques for HTML, tables, and matching.
Fix a generic ADAPT error	Testing and Debugging	Local rendering with line-level error messages.

If you want a quick vocabulary reset before you start, use [Common terms and names](#) for the common terms used throughout the book.

Apply it today

- Pick one page that matches what you are building today and copy-edit the example rather than starting from scratch.
- Keep regular PG minimal and write student-facing sentences in PGML.

[1.1: What is WeBWorK?](#)

[1.2: Why use WeBWorK?](#)

[1.3: Key Features of WeBWorK](#)

[1.4: Comparing WeBWorK to other formats](#)

[1.5: Common Terms and Names](#)

[1.6: Quickstart - Edit Your First Problem](#)

This page titled [1: Introduction](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

1.1: What is WeBWorK?

WeBWorK delivers automatically graded questions to students through a web interface. As an author, you write a text-based problem file (usually a `.pg` file) that defines values, a prompt, answer blanks, and grading rules.

A single problem can randomize values, present structured data, collect student answers, and grade them immediately. In this guide, regular PG is treated as the setup layer and PGML is treated as the student-facing prompt (see [Problem Generation \(PG\)](#) for more on PG).

This book targets the PG 2.17 subset used in ADAPT and webwork-pg-renderer, which means some macros found in full WeBWorK installs are not available. Keep that constraint in mind when you see online examples that load macros you do not recognize (see [Common terms and names](#) for common terms).

PGML-first rule of thumb

WeBWorK problems are written in a language called PG, but most authors should treat "regular PG" as the plumbing that sets up variables and answer checkers. The student-facing prompt should be written in PGML (PG Markup Language), which is designed to replace older `BEGIN_TEXT` authoring.

Rule of thumb: if it is a sentence the student reads, it belongs in PGML.

- PG (minimal): load macros, choose contexts, define random values, and build MathObjects.
- PGML (primary): write the prompt, math, tables, and answer blanks using readable markup.

You should be able to edit most biology questions in this book without fully understanding the PG setup code, as long as you keep changes inside the PGML block.

Why it works well for life sciences

- Short prompts with structured data (tables, concentrations, rates, and units).
- Algorithmic variants for repeated practice without reusing the same numbers.
- Clear answer checking via MathObjects (numeric, formulas, lists, and recognition checks).
- Clean prompts for common biology patterns (dilutions, fold change, and small data tables).

Apply it today

- Write one prompt sentence in PGML and move any student-facing text out of regular PG.
- Pick one biology-native pattern (a dilution, a rate, or a fold change) and randomize only the numbers.

Next step: read [Why use WeBWorK](#) to decide when a prompt belongs in WeBWorK and when a simpler tool is a better fit.

This page titled [1.1: What is WeBWorK?](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

1.2: Why use WeBWorK?

WeBWorK is useful when you want repeated, low-friction practice with immediate feedback and consistent grading. In science courses, the strength is not "more math", but rather repeatable workflows for concentrations, dilutions, rates, fold change, and small data tables inside realistic prompts.

Choose WeBWorK when

- You want repeated practice with immediate feedback and consistent grading.
- You want algorithmic variants where only the numbers change, not the story.
- You want explicit answer checking (tolerances, formats, and controlled text variants).
- You want prompts that mix text, values, and structured data without fragile formatting.

Choose something else when

- The task depends on rich interaction that is better served by a dedicated widget or simulation.
- The assessment is primarily writing or multi-step reasoning that is not a good fit for auto-grading.
- You need a standardized export/import workflow and accept the constraints of an LMS package format.

WeBWorK is a good fit when you want students to calculate, interpret, or compare values, but a poor fit for long written explanations or diagram drawing.

Why PGML matters

While a WeBWorK problem is "a PG file", this guide discourages writing long blocks of text in regular PG. PGML is designed so that the prompt reads like the rendered output, and answer blanks can be tied directly to answers in the same place you write the question.

- Readable prompts with inline and display math.
- Answer blanks like `[_____]{...}` that stay close to the text they grade.
- Variable substitution that formats MathObjects correctly in TeX or calculator notation.

Science-friendly patterns

- Use a short narrative plus a table of measurements.
- Ask for one computed quantity with clear units and controlled rounding.
- Include one or two recognition checks (multiple choice or matching) when appropriate (see [Different Question Types](#) for question types).
- Prefer small, constrained randomizations over complicated logic.
- Reuse a stable template so future edits stay small and predictable.

Tradeoffs to be honest about

- There is setup cost (macros, contexts, and answer checkers), especially for new authors.
- Randomization needs constraints and previewing, or you will generate broken variants.
- Format decisions (units, rounding, accepted text variants) must be stated to avoid grading disputes.

Apply it today

- Pick one life-science workflow (a dilution, a rate, or a fold change) and write the prompt in PGML.
- Write down the reporting rule (units and rounding) next to the answer blank.

How WeBWorK compares (briefly)

Other systems can also deliver automated questions (for example H5P interactive content, standardized LMS-import item packages, and LMS-native quiz questions). The main reason to choose WeBWorK in a science course is when you want robust answer parsing and checking (MathObjects), repeatable algorithmic variants, and clear, readable prompt authoring via PGML.

For a more detailed comparison and "choose this when" guidance, see [Comparing WeBWorK to other formats](#).

This page titled [1.2: Why use WeBWork?](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

1.3: Key Features of WeBWorK

WeBWorK's power is not a single feature, but a stable combination: algorithmic question setup, structured answer checking, and readable authoring of the student prompt.

What matters for authors

- PGML for prompt authoring (recommended): readable markup, math delimiters, and answer blanks (see [PGML \(PG Markup Language\)](#)).
- MathObjects for robust parsing and checking of student answers (numeric, formulas, lists).
- Randomization primitives (e.g., `random()`) to generate variants while keeping the scoring rules identical (see [Advanced Randomization Techniques](#)).
- Multiple answer interaction types via macros (numeric answers, formulas, radio buttons, etc.).

If you are new, focus first on PGML, numeric answer checking, and fixed-seed previewing (see [Testing and Debugging](#) for testing and debugging); everything else can wait.

Use this page as a checklist when you plan a new problem so you do not overbuild the setup.

What matters for life-science prompts

- Prompts that mix words, values, and math without fragile escaping.
- Tables for small datasets (measurements, lanes, conditions) in a readable format.
- Common biology workflows such as concentrations, dilutions, rates, and fold change.
- Clear expectations about units, rounding, and accepted formats to reduce grading disputes.

Tradeoffs and constraints

- There is setup cost (macros, contexts, and answer checkers), so templates matter.
- Randomization needs constraints and previewing to avoid broken variants.
- Auto-grading is only as clear as your reporting rules (units, rounding, accepted text variants).

Apply it today

- Pick the interaction first, then keep setup minimal and write the student-facing prompt in PGML.
- Write one biology-native task (a dilution, a rate, or a fold change) and decide the reporting rule up front.

Next step: use [Comparing WeBWorK to other formats](#) to choose WeBWorK versus another format for a specific activity.

This page titled [1.3: Key Features of WeBWorK](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

1.4: Comparing WeBWorK to other formats

Many platforms support automated or semi-automated assessment. The goal of this section is not to rank tools, but to help you choose the right tool for the learning outcome and the workflow you want for students and instructors.

Most life-science courses use more than one assessment tool; this book assumes WeBWorK is one piece of that ecosystem, not the entire solution.

Quick comparison

Quick comparison: format, best for, and tradeoffs.

Format	Best for	Tradeoffs
WeBWorK (PGML-first)	Robust answer checking, algorithmic variants, and science prompts that mix text, values, math, and tables.	Requires some author setup (macros, contexts, MathObjects); best results come from using templates and keeping setup small.
WeBWorK author workflow	Fast debugging when ADAPT is opaque, using local rendering for line-level errors.	Requires local renderer setup (Testing and Debugging) while still using ADAPT preview as the final check.
H5P interactive content	Highly interactive activities, quick engagement checks, and media-rich prompts with a low barrier to entry.	Answer checking is often pattern-based or limited to the widget type; algorithmic variants and strict parsing can be harder.
Standardized item packages	Portability across LMS systems and standardized quiz import/export workflows.	Question types and scoring can be constrained by the LMS; complex parsing or custom checks are often limited.
LMS-native question banks	Simple quizzes tightly integrated with grades, due dates, and course logistics.	Authoring is platform-specific; algorithmic and formula-based grading can be limited or inconsistent across platforms.
IMathAS / similar systems	Algorithmic questions with established patterns and large existing libraries.	Authoring conventions differ; migrating content between systems may require rewrites.

Why WeBWorK is a good fit for sciences

- **Science-style prompts:** short narrative plus explicit numbers, units, and often a small table of data.
- **Robust answer checking:** MathObjects parse student input and can enforce format, tolerances, and domain restrictions.
- **PGML for readability:** the prompt is written in a markup style that stays close to what students see, which helps maintainability.
- **Safe randomization:** randomize values while keeping the story and scoring rules stable.

Choose WeBWorK when

- You want multiple algorithmic versions of the same question.
- You want students to enter values or formulas in a structured way.
- You want to grade based on meaning, not only string matching.
- You want to reuse and version-control questions as text files.

Choose something else when

- The learning outcome depends on rich interaction that is better served by a dedicated widget (platform-specific drag-and-drop, hotspots, complex UI interactions).
- You need a standardized export/import workflow and are comfortable with the constraints of the target LMS.
- The assessment is primarily writing or multi-step reasoning where auto-grading is not the main goal.

A practical hybrid approach

Many courses mix tools:

- Use WeBWorK for numeric and formula-based practice with clear feedback.
- Use interactive activities for concept exploration or lab technique simulations.
- Use writing tools or projects for synthesis and communication skills.

This book focuses on WeBWorK with a PGML-first workflow. Treat regular PG as minimal setup code needed to support a clear PGML prompt.

Apply it today

- Pick one activity and decide whether you need robust parsing and grading, or whether a simpler format is enough.
- If you choose WeBWorK, start from a template and keep setup small.

Next step: use [Different Question Types](#) to pick an interaction pattern (multiple choice, matching, numerical entry, or fill-in-the-blank).

This page titled [1.4: Comparing WeBWorK to other formats](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

1.5: Common Terms and Names

This page defines the short list of terms you will see repeatedly in this book. The goal is to make the later chapters easier to read without stopping to decode vocabulary.

The terms are grouped into authoring terms and platform terms. Do not memorize these; return here when a term blocks your progress.

Common terms used in this book

Common terms used in this book reference.

Term	Meaning (in plain language)	Why you care while authoring
WeBWorK authoring	Terms you will see inside problem files.	These are the knobs you touch most often.
PG	The problem-file language that WeBWorK runs.	In this book, PG stays minimal and supports the PGML prompt.
PGML	A markup layer inside PG for writing the student prompt.	Most text, structure, and blanks should live here.
Macros	Reusable code libraries loaded by a problem file.	Macros enable question types, answer checkers, and PGML features.
MathObjects	Typed answer objects (numbers, formulas, lists) used for grading.	They make checking reliable and reduce string-matching errors.
Context	Rules that control what student input is allowed.	Wrong Context is a common cause of unexpected grading.
Answer checker	The rule used to decide whether an answer is correct.	Tolerances and formats usually live here.
Seed	A number that reproduces one randomized variant.	Seeds let you debug the exact failing version.
ADAPT and LibreTexts platform	Platform terms used in ADAPT and LibreTexts.	Useful when you navigate, share, or troubleshoot.
ADAPT	LibreTexts' assessment and adaptive learning system.	This is where you assign questions to students and manage delivery.
WeBWorK	An open-source homework system used heavily in math and science.	This book focuses on WeBWorK problems that run inside ADAPT.
OPL (Open Problem Library)	A large, shared library of WeBWorK problems maintained by the community.	Great for patterns and macro references, but it has little life-science content.
H5P	A framework for interactive questions used in ADAPT.	Useful for some prompts, but less suited to algorithmic parsing.
Commons	LibreTexts' library of shared pages and resources.	Where course content and shared assets are often stored.
Insight	LibreTexts' knowledge base and documentation hub.	Where platform help and policies usually live.
Studio	LibreTexts' repository for H5P questions and media.	Often where non-WeBWorK items are stored and shared.

For platform-wide definitions beyond this quick list, consult the LibreTexts glossary used by your campus.

Open Problem Library (OPL): value and gaps

The Open Problem Library (OPL) is a large public set of WeBWorK problems that is invaluable for learning common patterns, macro combinations, and interaction styles. It is also a reality check for what question types exist in the wild and how authors structure PG and PGML.

Our 2026-01-17 corpus audit of the full OPL found that the subject mix is dominated by math and that there are no life_sciences DBsubject tags, with bio-ish term hits showing up in only about 0.3% of files. That means the OPL is not a strong source of life-science problem content, even though it is a rich source of structure.

In this book, we use the OPL to:

- Spot reliable PGML patterns and macro combinations.
- Set realistic expectations for interaction types and widget choices.
- Build safer authoring checklists that avoid legacy patterns.

Apply it today

- When a term is unfamiliar, return here before you change macros or grading rules.
- Use a fixed seed to reproduce any issue you need to debug.

This page titled [1.5: Common Terms and Names](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

1.6: Quickstart - Edit Your First Problem

This quickstart is designed to give you a complete win early: take a working PGML-first problem, copy it, and make a small edit that still grades correctly.

Quickstart steps

1. Copy a working template problem (start from [Breaking Down the Components](#) if you have one available).
2. Change only the story text first, keeping the variable names and math intact.
3. Preview multiple variants to confirm the prompt still makes sense.
4. Then change the numbers or the formula, one small change at a time.

Do not start by changing variable names or answer objects; change text first.

A minimal PGML-first skeleton (for copy-editing)

The file below is intentionally complete and minimal: it includes macros, setup, a PGML prompt, and an answer blank tied directly to its answer object.

```
DOCUMENT( );

loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");

$a = random(2, 8, 1);
$b = random(3, 9, 1);
$ans = Real($a + $b);

BEGIN_PGML
In a lab notebook, two counts were recorded: [$a] and [$b].

What is the total count?

[ `Total = ` ] [_____]{ $ans }
END_PGML

ENDDOCUMENT( );
```

After you can copy-edit a problem like this without breaking grading, you are ready to expand into tables, multi-part prompts, and tighter answer checking.

Apply it today

- Copy a working file, change one small thing, and preview multiple variants before changing anything else.
- Keep student-facing sentences in PGML and keep regular PG focused on values and answer checking.

- After copy-editing, preview in ADAPT, and if ADAPT reports only a generic error, use [Testing and Debugging](#) (webwork-pg-renderer) to get a line-level error and fix it fast.

Next step: use [Problem Generation \(PG\)](#) to adopt a consistent file structure, then use [Breaking Down the Components](#) as your first reusable template.

This page titled [1.6: Quickstart - Edit Your First Problem](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

2: Problem Generation (PG)

This page only covers the PG you need to support a PGML prompt.

Start here (task first)

- I need to add metadata headers (OPL) -> [OPL Header and Metadata](#).
- I need a clean starter .pg skeleton -> [PG Problem Files](#).
- I need to know where setup ends and PGML begins -> [Sections within a PG question](#).
- My problem does not render, missing macro or context? -> [Common PG Macros](#).
- I copied code from the internet and it looks weird -> [Legacy PG and deprecated patterns](#).
- I see a feature online that does not work in ADAPT -> [Future PG Version Features](#).

If you are copy-editing a working problem, you can skip this chapter and come back only when you need to change setup behavior.

WebWork problems are written in PG (see [Introduction to PG Language](#) for an introduction), but this guide is not trying to turn instructors into Perl developers. Regular PG is treated as the minimum setup needed to support a clean PGML prompt: define a few variables, define answer checkers, and let PGML carry the student-facing text.

Library-quality problems also need an OPL header section that comes before `DOCUMENT( )`, so the five-section reality is: OPL header, Preamble, Setup, Statement, and Solution.

What PG is used for in this guide

- Load the macros needed for PGML and answer checking.
- Choose a MathObjects context that matches the expected input.
- Generate a small number of randomized values with sensible ranges.
- Compute the target answer(s) and attach the right answer checkers.

What regular PG is not used for in this guide

- Writing the prompt as long blocks of `BEGIN_TEXT/END_TEXT`.
- Building complex formatting by hand (PGML is preferred).
- Turning a single problem into a large Perl script.

If a problem feels like it needs complexity, prefer clearer prompts, tighter ranges, or a different interaction pattern before adding more PG logic.

How this chapter fits the book

Chapter 2 exists because every PGML problem still needs some PG scaffolding ([macros](#), variables, answers). The goal is to keep that scaffolding short, readable, and reusable.

[2.1: Introduction to PG Language](#)

[2.2: OPL Header and Metadata](#)

[2.3: PG Problem Files with a .pg file extension](#)

[2.4: Sections within a PG question](#)

[2.5: Common PG Macros](#)

[2.6: Legacy PG and deprecated patterns](#)

[2.7: Future PG version features](#)

This page titled [2: Problem Generation \(PG\)](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.1: Introduction to PG Language

PG is the programming language behind every WeBWorK problem. It is implemented in Perl and provides macros for answer checking, randomization, and formatting. Think of PG as the scaffolding that supports the student-facing prompt: it loads the tools you need, sets up variables, and wires the grading, but the text the student actually reads belongs in PGML.

This page covers only the PG you need to support a PGML prompt. If you can read this page, you already know enough PG to write most science and math problems. Set up a local renderer to test your work ([Setting Up the PG Renderer](#)).

Use this page when: you need to change setup behavior without touching the student-facing prompt.

The five-component problem structure

Every PG problem file follows the same five components, in order. Keeping this structure consistent makes problems easier to read, debug, and share.

1. **OPL header:** metadata comments for searchability and attribution (covered in [OPL Header and Metadata](#); see also [Breaking Down the Components](#) for a detailed component breakdown).
2. **Preamble:** `DOCUMENT();` and `loadMacros(...)`.
3. **Setup:** context selection, random values, computed answers, and answer checkers.
4. **Statement:** the PGML text block that the student reads and interacts with (see [PGML \(PG Markup Language\)](#) for PGML details).
5. **Solution:** a solution block (and optional hints) for student review.

Canonical problem skeleton

Copy this skeleton and fill in the pieces you need.

```
## (Tagging header block goes here -- see Section 2.2)

DOCUMENT();

# -----
# 1) Preamble
# -----
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

$showPartialCorrectAnswers = 1;

# -----
# 2) Setup
# -----
Context("Numeric");
$a = random(2, 9, 1);
$b = random(3, 9, 1);
$ans = Compute("$a + $b");
```

```
# -----
# 3) Statement (PGML)
# -----

BEGIN_PGML
Compute  $[a] + [b]$ .

Answer:  $[ \rule{1cm}{0.4pt} ] \{ \$ans \}$ 
END_PGML

# -----
# 4) Solution
# -----

BEGIN_PGML_SOLUTION
The answer is  $[a] + [b] = [ans]$ .
END_PGML_SOLUTION

ENDDOCUMENT();
```

Key rules

- Every file must start with `DOCUMENT();` and end with `ENDDOCUMENT();` .
- Put all macro loads in a single `loadMacros(...)` block in the preamble.
- All student-facing text goes inside `BEGIN_PGML / END_PGML` blocks. Do not write long prompts in raw PG.
- Always include a `BEGIN_PGML_SOLUTION ... END_PGML_SOLUTION` block, even if it is only a single sentence.
- Keep the five sections in order and visually separated with comment headings.

Minimum recommended macros

Start every problem with these four macros. Add others only when you need them.

Minimum recommended macros: macro and purpose.

Macro	Purpose
PGstandard.pl	Core PG functions; required in practice for every problem.
MathObjects.pl	Modern answer types such as <code>Real()</code> , <code>Compute()</code> , and <code>Formula()</code> .
PGML.pl	Enables <code>BEGIN_PGML</code> / <code>END_PGML</code> blocks for writing prompts.
PGcourse.pl	Commonly expected by WeBWorK installations; include unless you have a reason to drop it.

If you add a specialized macro (for example `parserPopUp.pl` for dropdowns), add a short comment explaining why.

PGML-first authoring

In PGML-first problems, the student-facing text and answer blanks live entirely inside PGML blocks. PG handles everything that happens before and after the prompt: loading macros, choosing a context, generating random values, and defining correct answers.

Inline grading principle: attach answer evaluators directly to blanks in the PGML block. Do not create a blank in PGML and then wire the grader separately with `ANS( ... )` calls.

Preferred: evaluator attached at the blank
BEGIN_PGML
Answer: [_____]{\\$ans}

```
END_PGML

# Avoid: blank in PGML with separate ANS() wiring
BEGIN_PGML
Answer: [_____]
END_PGML
ANS($ans->cmp());
```

When the evaluator lives next to the blank, the connection between the prompt and the grading stays obvious. This is especially helpful for problems with multiple answer blanks.

File extensions: .pgml vs .pg

Use the `.pgml` extension for full PGML-first problems where the statement and graders are entirely inside PGML blocks. If a problem uses legacy `ANS(...)` wiring or mixed PG/PGML structure, keep the extension as `.pg`. The extension signals whether tooling should expect pure PGML structure.

Context selection

A context controls what kind of input WeBWorK accepts from the student. Always set an explicit context in the Setup section before defining answers.

Context selection: context and when to use.

Context	When to use
Numeric	General numeric answers (the most common default).
Real	Answers that must be real numbers (no complex values).
Vector	Answers that are vectors.
Matrix	Answers that are matrices.
Complex	Answers that involve complex numbers.

Question order recommendation

Inside the PGML block, present information in this order so that questions are consistent and scannable for students.

1. **Background:** context or scenario the student needs to understand.
2. **Statement:** what is being asked.
3. **Data:** any tables or given values.
4. **Directions:** how to respond, constraints, formatting expectations.

Keep directions concise and place them after the data or statement rather than before the prompt.

PG concepts you actually need

PG concepts you actually need: pg concept, why you care, and what you type.

PG concept	Why you care	What you type
Macro loading	Enables PGML and answer checking.	<code>loadMacros(...)</code>
Context	Controls what student input is allowed.	<code>Context("Numeric")</code>
Random values	Creates variants without changing the story.	<code>random(2, 8, 1)</code>
MathObjects	Builds answers the grader can check.	<code>Real(...)</code> or <code>Compute(...)</code>
Answer checker	Defines how grading works.	<code>\$ans->cmp()</code>

Common failure and fix

```
Broken: $ans = Real($a + $b);  
What you will see: Undefined subroutine &main::Real called  
Fix: load "MathObjects.pl" before using Real(...)
```

A mindset for science problems

- Randomize the values, not the story.
- Keep units explicit and consistent (and prevent unit drift: μM vs mM , μL vs mL).
- Use realistic ranges (no negative concentrations, no 10,000 μL pipettes).
- For text answers, state naming expectations (gene/protein symbols, case, and hyphens).
- Prefer one or two answer blanks with clear interpretation.
- When you need a table or structured layout, reach for PGML helpers rather than ad-hoc HTML.

Safe defaults (micro-pattern)

- Choose a context early.
- Define a small set of values with constrained ranges.
- Define answers as MathObjects.
- Attach answers directly to blanks in PGML.

You can stop here if you are only editing prompts or adding one numeric value. This page is enough to keep your PG setup stable.

Next, depending on what you are doing:

- Need the OPL header? Go to [OPL Header and Metadata](#).
- Need a clean starter .pg skeleton? Go to [PG Problem Files](#).
- Not sure where setup ends and PGML begins? Go to [Sections within a PG question](#).
- Need macro help? Go to [Common PG Macros](#).

This page titled [2.1: Introduction to PG Language](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.2: OPL Header and Metadata

The OPL header is the metadata block that comes before `DOCUMENT()` and makes a problem searchable, classifiable, and reusable. In this guide, the header is treated as a real section ("section zero") even though traditional PG documentation only lists four sections.

Use this page when: you are creating a new problem or adapting an existing one and need correct metadata for long-term reuse.

Use the templates in [Minimal Templates](#) as your starting point so you do not forget required fields.

Required and recommended fields

Required and recommended fields: field, status, and why it matters.

Field	Status	Why it matters
<code>TITLE('...')</code>	Required	Short, student-facing title used in search and listings.
<code>DESCRIPTION</code> block	Required	Two to three sentences describing the task and constraints.
<code>KEYWORDS(...)</code>	Strongly recommended	Improves search and grouping across a bank.
<code>DBsubject/DBchapter/DBsection</code>	Strongly recommended	Primary classification tags for the Open Problem Library.
<code>Level(...)</code>	Required	Bloom's taxonomy level (1-6) for difficulty classification.
Attribution fields	Recommended	Credits authors and institutions for long-term reuse.
<code>Language(en)</code>	Required	ISO 639-1 code specifying the problem language.

Copy and edit (OPL header template)

```
## TITLE('Short descriptive title')
## DESCRIPTION
## One to three sentences describing the task and constraints.
## ENDDescription
## KEYWORDS('keyword1','keyword2','keyword3')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Kinetics basics')
## Level(2)
## Date('2026-01-28')
## Author('Author name')
## Institution('Institution name')
## Language(en)
```

TITLE field guidance

The TITLE is the student-facing name that appears in problem listings and search results. Keep it short (under 80 characters), descriptive, and focused on the task rather than the story wrapper.

Good vs bad titles

Good vs bad titles: good title, bad title, and why.

Good title	Bad title	Why
Compute dilution factor from stock and final concentration	This is a problem about dilutions in the lab	Bad: filler words, says "problem", not task-focused
X-linked inheritance from fly offspring counts	Genetics practice	Bad: vague, no detail, "practice" is filler
Enzyme kinetics: calculate K_m from Lineweaver-Burk	Calculate Michaelis constant using the reciprocal transformation approach	Bad: too long (> 80 chars)
Determine qPCR fold change from Ct values	qPCR	Bad: too terse, doesn't describe the task

DESCRIPTION block guidance

The DESCRIPTION block is a 2-3 sentence instructor-facing summary that explains the task, any constraints that affect adoption (unit requirements, rounding, domain restrictions), and provenance if adapted from another source.

Template with placeholders

```
## DESCRIPTION
## [Task in one sentence]. [Constraints or special requirements]. [Optional: Adapted from source]
## ENDDescription
```

Biology examples of complete DESCRIPTION blocks

Enzyme kinetics problem

```
## DESCRIPTION
## Students calculate the Michaelis constant ( $K_m$ ) from enzyme kinetics data using a
## Lineweaver-Burk plot. Answers require two significant figures and appropriate units.
## ENDDescription
```

Genetics cross problem

```
## DESCRIPTION
## Students determine the probability of offspring genotypes from a dihybrid cross using
## Punnett square logic. Answers are fractions in lowest terms (e.g., 9/16).
## ENDDescription
```

Dilution series problem

```
## DESCRIPTION
## Students calculate final concentrations in a serial dilution series given a stock
## concentration and dilution factors. Answers must include units ( $\mu\text{M}$  or  $\text{mM}$ ) and use
## scientific notation for very small values.
## ENDDescription
```

What NOT to include in DESCRIPTION

- Do not paste full solutions or step-by-step instructions.
- Do not include long pedagogical essays or teaching philosophy.
- Do not include code or implementation details.
- Do not describe the story wrapper (e.g., "students imagine they are in a lab").

KEYWORDS guidance

Use 3 to 5 keywords (prefer 3-4 when possible) that describe techniques, concepts, or methods. Avoid character names, story nouns, or terms already implied by DBsubject.

Keyword selection process for biology problems

Example 1: Dilution problem

- Good keywords: 'serial dilution', 'concentration', 'stock solution'
- Bad keywords: 'biology', 'lab', 'science', 'tube'
- Why: Good keywords are searchable techniques. Bad keywords are too generic or story nouns.

Example 2: Enzyme kinetics problem

- Good keywords: 'Michaelis-Menten', 'Lineweaver-Burk', 'enzyme kinetics', 'Km'
- Bad keywords: 'biochemistry', 'enzyme', 'calculation'
- Why: Good keywords name specific methods. Bad keywords duplicate DBsubject or are too broad.

Example 3: qPCR problem

- Good keywords: 'qPCR', 'fold change', 'Ct value', 'delta-delta Ct'
- Bad keywords: 'gene expression', 'RNA', 'technique'
- Why: Good keywords are technique-specific. Bad keywords are outcome terms, not methods.

How to avoid duplicating DBsubject terms

If DBsubject is Biochemistry, do not use 'biochemistry' as a keyword. Instead, use technique names like 'enzyme kinetics', 'Henderson-Hasselbalch', or 'buffer capacity'. Keywords should narrow the search within the subject area.

DBsubject, DBchapter, and DBsection guidance

These three fields form the primary classification taxonomy for the Open Problem Library. Choose values that fit life science courses and match existing usage in your repository for consistency.

Biology-appropriate DBsubject values

Biology-appropriate DBsubject values: dbsubject value and when to use.

DBsubject value	When to use
Biology - molecular	DNA, RNA, transcription, translation, replication
Biology - introductory	General biology concepts, cell structure, basic processes
Biochemistry	Enzyme kinetics, metabolism, protein structure, buffers
Genetics	Inheritance patterns, Punnett squares, linkage, pedigrees
Laboratory Techniques	Dilutions, spectrophotometry, gel electrophoresis, qPCR
Cell Biology	Organelles, transport, signaling, cell cycle
Biostatistics	Statistical tests, experimental design, data analysis

Decision tree: choosing the right DBsubject

1. **What is the primary discipline?** Start with the course context (Biochemistry, Genetics, etc.)
2. **Is it a technique or concept?** If purely technique-focused (dilutions, PCR), use Laboratory Techniques.
3. **Is it broad or specific?** If the problem spans multiple sub-disciplines, use the most specific subject that still fits typical course organization.
4. **Does it match existing usage?** Search your repository for similar problems and copy the exact DBsubject string to maintain consistency.

Full list of biology-appropriate DBsubject values

Use these DBsubject values for life science content. If a new subject is needed, add it to the list so tags remain consistent across the repository.

Full list of biology-appropriate DBsubject values reference.

DBsubject value	When to use
Anatomy and Physiology	Organ systems, body structures, physiological processes
Biochemistry	Enzyme kinetics, metabolism, protein structure, buffers
Bioinformatics and Computational Biology	Sequence analysis, phylogenetics, computational methods
Biomedical Sciences	Disease mechanisms, pharmacology, clinical applications
Biostatistics	Statistical tests, experimental design, data analysis
Cell Biology	Organelles, transport, signaling, cell cycle
DNA Profiling	Forensic DNA analysis, STR markers, electrophoresis patterns
Ecology	Population dynamics, ecosystems, biodiversity
Evolution	Natural selection, phylogenetics, population genetics
Immunology	Immune responses, antibodies, vaccines, immunoassays
Inheritance Genetics	Inheritance patterns, Punnett squares, linkage, pedigrees
Laboratory Techniques	Dilutions, spectrophotometry, gel electrophoresis, qPCR
Microbiology	Bacteria, viruses, microbial growth, antimicrobials
Molecular Biology	DNA replication, transcription, translation, gene regulation
Neuroscience	Neural signaling, brain structure, synaptic transmission
Organismic Biology	Whole-organism biology, plant biology, animal biology

Complete classification examples

Example 1: Dilution problem

```
## DBsubject('Laboratory Techniques')
## DBchapter('Solution preparation')
## DBsection('Serial dilutions')
```

Example 2: Enzyme kinetics problem

```
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Kinetics basics')
```

Example 3: Genetics cross problem

```
## DBsubject('Genetics')
## DBchapter('Mendelian inheritance')
## DBsection('Dihybrid crosses')
```

Problem level: Bloom's taxonomy scale

The `Level(...)` field uses a 1-6 scale aligned to Bloom's revised taxonomy: Remember, Understand, Apply, Analyze, Evaluate, Create. This field is required for problems in the biology-problems repository.

Problem level: Bloom's taxonomy scale: level, bloom's category, verbs, and more.

Level	Bloom's category	Verbs	Biology example
1	Remember	recall, define, label, list, match, identify	"Which amino acid has a sulfur atom in its side chain?"
2	Understand	explain, summarize, interpret, classify, compare	"Classify this molecule as a carbohydrate, lipid, protein, or nucleic acid."
3	Apply	apply, use, determine, solve, choose	"Using the genetic code table, translate this mRNA sequence."
4	Analyze	analyze, compare, contrast, categorize, infer	"Analyze this gel electrophoresis result to determine which protein is largest."
5	Evaluate	evaluate, justify, critique, defend	"Justify why an Arg-Glu-Asp peptide would be more soluble than Leu-Val-Ile at pH 7.0."
6	Create	design, propose, formulate, develop, create	"Design a peptide sequence that would be maximally soluble at pH 7.0."

Level assessment tips

- **Level 1 (Remember):** Direct recall, no inference or multi-step reasoning needed. Asking for definitions, names, or lists.
- **Level 2 (Understand):** Requires understanding and classification, but necessary reference values are given.
- **Level 3 (Apply):** Direct application of rules or formulas with clear instructions and provided data.
- **Level 4 (Analyze):** Student must infer relationships, analyze multiple factors, or synthesize information from different knowledge domains.
- **Level 5 (Evaluate):** Requires defending a conclusion, evaluating competing explanations, or justifying choices based on multiple criteria.
- **Level 6 (Create):** Requires synthesizing knowledge to create novel solutions, designs, or models.

Level format

```
## Level(3)
```

Language field

Use ISO 639-1 language codes to specify the problem language. This field is required. For English, use:

```
## Language(en)
```

Attribution and provenance fields

These fields credit authors, track provenance, and clarify licensing. They are recommended for all problems and required for adapted content.

Minimal recommended set

```
## Date('2026-01-28')
## Author('Dr. Neil R. Voss')
## Institution('Roosevelt University')
```

Rules for attribution

- If the problem is adapted, also state that in DESCRIPTION (e.g., "Adapted from ...").
- Use a real date when known. Prefer ISO format: YYYY-MM-DD.
- Empty strings are acceptable when you do not know the details, but do not invent information.

Textbook provenance fields

Use these when the problem is adapted from a textbook, worksheet, or other source. Number the tags (TitleText1, TitleText2, etc.) to map to multiple texts when appropriate.

```
## TitleText1('Title of source')
## EditionText1('Edition')
## AuthorText1('Author(s)')
## Section1('Section identifier')
## Problem1('Problem identifier')
```

License comments

OPL does not define a standard license tag. Use plain comment lines (single # , not ##) to state licensing for the problem text and any included images or code.

```
# This work is licensed under CC BY 4.0 (Creative Commons Attribution 4.0
# International License).
# https://creativecommons.org/licenses/by/4.0/
# Source code portions are licensed under LGPLv3.
```

Tagging quality rules

1. DBsubject reflects the discipline and course area, not the scenario.
2. KEYWORDS reflect methods and concepts, not superficial nouns.
3. Use consistent capitalization and spelling across the corpus.
4. Prefer stable, existing label strings over "better" new strings.
5. Avoid "meta noise" labels like `WeBWoRk` or `ZZZ-Inserted Text` in new content.

Header review checklist

Before publishing a problem, verify the following:

- DESCRIPTION exists and is short, accurate, and code-free.
- KEYWORDS exist and are concept-focused (3-5 terms).
- DBsubject uses an existing repo string from the local list.
- DBchapter and DBsection are present (no empty strings).
- Level is set using the 1-6 Bloom's taxonomy scale.
- Author and Date are present when known.
- Language(en) is present.
- No smart quotes or non-ASCII characters in header lines.

Complete worked examples

These examples show full OPL headers for different problem types. Copy and adapt for your own problems. For more templates, see [Minimal Templates](#).

Full header for a dilution problem

```
## TITLE('Serial dilution: calculate final concentrations')
## DESCRIPTION
## Students calculate final concentrations in a serial dilution series given a stock
## concentration and dilution factors. Answers must include units ( $\mu\text{M}$  or  $\text{mM}$ ) and use
## scientific notation for very small values.
## ENDDescription
## KEYWORDS('serial dilution','concentration','stock solution')
## DBsubject('Laboratory Techniques')
## DBchapter('Solution preparation')
## DBsection('Serial dilutions')
## Level(3)
## Date('2026-01-28')
## Author('Dr. Neil R. Voss')
## Institution('Roosevelt University')
## Language(en)
```

Full header for a genetics problem

```
## TITLE('Dihybrid cross: predict offspring ratios')
## DESCRIPTION
## Students determine the probability of offspring genotypes from a dihybrid cross using
## Punnett square logic. Answers are fractions in lowest terms (e.g., 9/16).
## ENDDescription
## KEYWORDS('dihybrid cross','Punnett square','Mendelian inheritance','independent assortment')
## DBsubject('Genetics')
## DBchapter('Mendelian inheritance')
## DBsection('Dihybrid crosses')
## Level(3)
## Date('2026-01-28')
## Author('Dr. Neil R. Voss')
## Institution('Roosevelt University')
## Language(en)
```

Full header for a pathway problem

```
## TITLE('Glycolysis: identify rate-limiting enzymes')
## DESCRIPTION
## Students identify which enzymes in the glycolytic pathway are subject to allosteric
## regulation and explain their physiological significance. Free-response answers are
## graded for keyword matches.
## ENDDescription
## KEYWORDS('glycolysis','allosteric regulation','rate-limiting enzyme','metabolism')
## DBsubject('Biochemistry')
## DBchapter('Metabolism')
## DBsection('Glycolysis')
```

```
## Level(4)
## Date('2026-01-28')
## Author('Dr. Neil R. Voss')
## Institution('Roosevelt University')
## Language(en)
```

Common mistakes and how to avoid them

- **Empty DBsubject string:** Never leave DBsubject, DBchapter, or DBsection empty. Choose the closest existing label and revise later if needed.
- **Smart quotes in header:** Use plain ASCII quotes ('), not curly quotes (' or '). Smart quotes break OPL parsing tools.
- **Missing ENDDescription tag:** Always close the DESCRIPTION block with `## ENDDescription` . Without it, parsers may read subsequent code as part of the description.
- **Too many keywords:** More than 5 keywords dilutes search effectiveness. Stick to 3-5, preferring the most distinctive terms.
- **Using "practice" or "problem" in title:** Titles should describe the task, not announce that it's a practice problem. Replace "Practice computing dilutions" with "Compute dilution factor from concentrations".
- **Pasting code into DESCRIPTION:** The description is for humans, not parsers. Keep it plain English and code-free.

Apply it today

- Before editing a problem, confirm it has a TITLE and DESCRIPTION block.
- Copy the template from [Minimal Templates](#) and fill in metadata before writing PGML.

This page titled [2.2: OPL Header and Metadata](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.3: PG Problem Files with a .pg file extension

This page only covers the PG you need to support a PGML prompt.

Use this page when starting a new file or diagnosing a broken one.

Use this page when: you need a clean, reliable .pg skeleton you can reuse.

2.3.1 Copy and edit (PGML-first skeleton)

```
# ===== OPL HEADER =====
## TITLE('Short descriptive title')
## DESCRIPTION
## One to three sentences describing the task and constraints.
## ENDDescription
## KEYWORDS('keyword1','keyword2','keyword3')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Kinetics basics')
## Date('2026-01-28')
## Author('Author name')
## Institution('Institution name')

# ===== PREAMBLE =====
DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

# ===== SETUP =====
Context("Numeric");
$a = random(2, 8, 1);
$b = random(3, 9, 1);
$ans = Real($a + $b);

# ===== STATEMENT =====
BEGIN_PGML
[ `Total = ` ] [ ____ ] { $ans }
END_PGML

# ===== SOLUTION =====
# BEGIN_PGML_SOLUTION
# END_PGML_SOLUTION

ENDDOCUMENT();
```

Copy and edit (PGML-first skeleton): change this first and do not touch yet.

Change this first	Do not touch yet
• Prompt text in PGML • Random ranges and values	• Macro list and order • DOCUMENT()/ENDDOCUMENT()

2.3.1.1 Common failure and fix

Broken: BEGIN_PGML before DOCUMENT();
 What you will see: PGML block not recognized or raw text output
 Fix: Keep DOCUMENT() and loadMacros() above PGML

After any macro change, render locally to catch missing macros early ([Testing and Debugging](#)).

A WeBWorK problem is typically stored in a file with a .pg extension. The file is a mix of:

- Comments (for authors)
- PG code (setup and answer checking)
- PGML text blocks (the student-facing prompt)

2.3.2 Common structure (five sections)

Most .pg files follow a predictable structure. When you know the structure, you can read problems faster and keep your own authoring consistent.

1. **OPL header:** metadata comments for searchability and attribution.
2. **Preamble:** DOCUMENT(); and loadMacros(...); (see [Preamble](#)).
3. **Setup:** contexts, variables, randomization, answer objects.
4. **Statement:** the student-facing prompt, typically in BEGIN_PGML .
5. **Solution (optional):** hints/solutions and any remaining answer setup. With PGML-first authoring, answers are usually attached directly to blanks, so a separate answer section is often unnecessary.

File structure errors are the most common cause of total failure. Most bugs at this level are mechanical, not conceptual.

2.3.3 Most common failure modes

- Missing DOCUMENT(); or ENDDOCUMENT(); .
- A macro is loaded after it is needed.
- A PGML block appears outside the document lifecycle.

2.3.4 Annotated skeleton (recommended)

When you are learning, it helps to keep a short commented skeleton and copy-edit it rather than starting from a blank file.

```
# 1) OPL header (recommended)
## TITLE('Short descriptive title')
## DESCRIPTION
## ... summary for future authors ...
## ENDDescription
## KEYWORDS('keyword1','keyword2','keyword3')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Kinetics basics')
## Date('2026-01-28')
## Author('Author name')
```

```
## Institution('Institution name')

# 2) Initialization (required)
DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

# 3) Setup (keep small)
Context("Numeric");
# ... choose random values ...
# ... define answer objects and graders ...

# 4) Text (student-facing prompt)
BEGIN_PGML
... prompt text ...
... answer blanks tied to answers ...
END_PGML

# 5) Optional hint/solution blocks
# BEGIN_PGML_HINT ... END_PGML_HINT
# BEGIN_PGML_SOLUTION ... END_PGML_SOLUTION

ENDDOCUMENT();
```

2.3.5 What to keep stable

- Always include `DOCUMENT();` and `ENDDOCUMENT();` .
- Always load `PGML.pl` when you use `BEGIN_PGML` .
- Keep setup code short and readable; your future self will thank you.

2.3.6 Test immediately (recommended)

- After any macro or setup change, render locally with `webwork-pg-renderer` ([Testing and Debugging](#)) to catch missing macros and syntax errors early.
- If local setup feels like too much, preview in ADAPT instead, but expect slower debugging when something breaks.

2.3.6.1 Next, depending on what you are doing:

- Need macro help? Go to [Common PG Macros](#).
- Copied legacy code? Go to [Legacy PG and deprecated patterns](#).
- Not sure where setup ends and PGML begins? Go to [Sections within a PG question](#).

This page titled [2.3: PG Problem Files with a .pg file extension](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.4: Sections within a PG question

This page only covers the PG you need to support a PGML prompt.

Use this page when: you need to decide where a change belongs in the file.

Copy and edit (section map)

The block below is a complete PG file skeleton with all five sections labeled so you can see the overall structure at a glance.

```
# ===== OPL HEADER =====
## TITLE('Short descriptive title')
## DESCRIPTION
## One to three sentences describing the task and constraints.
## ENDDescription
## KEYWORDS('keyword1','keyword2','keyword3')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Kinetics basics')
## Date('2026-01-28')
## Author('Author name')
## Institution('Institution name')

# ===== PREAMBLE =====
DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

# ===== SETUP =====
Context("Numeric");
$a = random(2, 8, 1);
$ans = Real($a + 1);

# ===== STATEMENT =====
BEGIN_PGML
[ `Result = ` ] [ ____ ] { $ans }
END_PGML

# ===== SOLUTION =====
# BEGIN_PGML_SOLUTION
# END_PGML_SOLUTION

ENDDOCUMENT();
```

To adapt this skeleton, replace the OPL header fields with your own metadata, change the Context to match your answer type, set up your random values, and build your answer expression.

PG executes sections top-to-bottom, and the PGML block captures variable references at parse time. If you place `Context()` or answer object definitions after `BEGIN_PGML`, PGML will not see them because they do not exist yet when the block is parsed.

Common failure and fix

Broken: `Context("Numeric")` after `BEGIN_PGML`
 What you will see: parsing errors or wrong answer checking
 Fix: keep `Context` and answer objects in Setup above PGML

Where does this go?

Use the following table as a quick lookup for deciding which section a new line of code belongs in.

Where does this go?: task, section, and example.

Task	Section	Example
Tag the problem	OPL header	<code>## TITLE('...')</code>
Load macros	Preamble	<code>loadMacros(...)</code>
Choose a context	Setup	<code>Context("Numeric")</code>
Random values	Setup	<code>\$a = random(2, 8, 1);</code>
Attach an answer checker	Setup	<code>\$ans = Real(...);</code>
Show the prompt text	Statement	<code>BEGIN_PGML ... END_PGML</code>
Show a hint or solution	Solution	<code>BEGIN_PGML_SOLUTION</code>

Most WeBWorK examples describe a PG file as having several sections (description, initialization, setup, text, answers/solutions). That model is still useful, but when you write PGML-first (see [Breaking Down the Components](#) for a detailed breakdown), the roles shift:

Recommended sections (PGML-first)

1. **OPL header:** metadata comments for searchability and attribution.
2. **Preamble:** `DOCUMENT();` and `loadMacros(...)` .
3. **Setup:** define variables and answers, pick a context, set tolerances.
4. **Statement (PGML):** the prompt between `BEGIN_PGML` and `END_PGML` (see [Introduction to PGML](#) for PGML syntax).
5. **Solution (optional):** use `BEGIN_PGML_SOLUTION` or `BEGIN_PGML_HINT` when you want a structured solution.
6. **End:** `ENDDOCUMENT();` .

Why answers often live inside the PGML text

When you attach the correct answer directly to an answer blank (for example `[_____]{Real(12)}`), the connection between prompt and grading stays obvious. This is especially helpful for science prompts with multiple values and multiple parts.

When answers live next to the text they grade, it is much easier to see why a student got something wrong.

Rule of thumb: if it is a sentence the student reads, it belongs in PGML.

Next, depending on what you are doing:

- Need the OPL header? Go to [OPL Header and Metadata](#).
- Need a clean starter .pg skeleton? Go to [PG Problem Files](#).

- Need macro help? Go to [Common PG Macros](#).
- Copied legacy code? Go to [Legacy PG and deprecated patterns](#).

This page titled [2.4: Sections within a PG question](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.5: Common PG Macros

This page is not a catalog of everything PG can do. It is a short list of macros you will actually see and use when writing PGML-first science problems.

Use this page when: a problem will not render, a widget is missing, or you are adding a new interaction type.

This guide reflects the PG 2.17 subset used in ADAPT and webwork-pg-renderer. Some macros used in online examples are missing in this subset, so check the allowlist before you add a new file to `loadMacros()`.

For the full comparison table, see `docs/PG_MACRO_VERSION_COMPARISON.md`.

If you are writing your first problem, start with the four core macros (PGstandard.pl, MathObjects.pl, PGML.pl, PGcourse.pl). Add interaction macros only when you need a specific widget like radio buttons or dropdowns.

Copy and edit (minimal macro set)

```
DOCUMENT();

loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);
```

This is the most common macro error: calling a function that requires a macro you forgot to load. PG reports the function name as "Undefined subroutine" because the macro file that defines it was never loaded.

Common failure and fix

```
Broken: RadioButtons(...) without parserRadioButtons.pl
What you will see: Undefined subroutine &main::RadioButtons called
Fix: add "parserRadioButtons.pl" to loadMacros()
```

Core macros for PGML-first problems

These four macros form the foundation of every PGML problem. Load all four unless you have a specific reason to omit one. The table below shows what each macro enables and what breaks if you leave it out.

Core macros for PGML-first problems reference.

Macro	What it enables	What breaks if it is missing
PGstandard.pl	Core PG helpers and defaults.	Basic PG functions are undefined.
PGML.pl	PGML blocks, formatting, and answer blanks.	PGML blocks fail or render as raw text.
PGcourse.pl	Course-level helpers and defaults.	Course-specific helpers are undefined.
MathObjects.pl	MathObjects: Real, Formula, Compute, Context.	Needed when using Real(), Compute(), Formula(), or Context(). Not required for RadioButtons or CheckboxList.

Note: MathObjects.pl is included in most science problems because they use numeric answers with Real() or Compute(). For pure multiple-choice or matching problems that do not compute numeric values, it can be omitted.

Add-ons by interaction type

Beyond the core four, you add macros based on the interaction type your problem needs. Each row below corresponds to a student-facing widget. Load the macro only when you use that widget -- extra macros add no value and slow down debugging.

Add-ons by interaction type: what students see, macro to load, and notes.

What students see	Macro to load	Notes
Single-choice questions (radio buttons)	parserRadioButtons.pl	Visible options, one correct choice (see Multiple Choice).
Matching or shuffled lists	PGchoicemacros.pl	Matching and choice helpers (see Matching and Randomized Matching Problems).
Targeted feedback for common errors	answerHints.pl	Instructor-style feedback without extra widgets.
Dropdown menus (select one from a list)	parserPopUp.pl	Use PopUp() for inline dropdown selection. Available in ADAPT (see Randomized Matching Problems).
Drag-and-drop proof ordering	draggableProof.pl	Students arrange steps into the correct order.
Drag-and-drop grouping into subsets	draggableSubsets.pl	Students sort items into labeled bins.
Free-response essay box	PGessaymacros.pl	Open-ended text entry; requires manual grading.
Fill-in-the-blank word completion	parserWordCompletion.pl	Accepts specific string answers from a word bank.

Macro availability (subset vs full PG)

Not every PG macro is available in the ADAPT/renderer subset. The table below shows which macros work in your environment. If a macro shows "No" in the ADAPT column, you need a workaround.

Macro availability (subset vs full PG) reference.

Macro file	ADAPT/renderer subset	Full PG 2.17+	PG 2.20
PGML.pl	Yes	Yes	Yes
MathObjects.pl	Yes	Yes	Yes
parserRadioButtons.pl	Yes	Yes	Yes
parserCheckboxList.pl	No	Yes	Yes
VectorListCheckers.pl	No	Yes	Yes

What macro availability means for you

The ADAPT/renderer subset is based on PG 2.17 but uses a flattened macro tree with some files removed. This affects what widgets and helpers you can use in problems.

Parser widgets: Key differences

Parser widgets: Key differences reference.

Widget	ADAPT/renderer	Full PG 2.17+	Notes
PopUp	Yes	Yes	Use for matching dropdowns (see Randomized Matching Problems)
RadioButtons	Yes	Yes	Use for single-choice questions and per-statement True/False
CheckboxList	No	Yes	Not available. Use RadioButtons per statement instead (Randomized MC True and False Statements)
DropDown	No	No	PG 2.18+ only. Use PopUp in ADAPT/renderer

Core macros: All available

Core macros: All available: macro, adapt/renderer, full pg 2.17+, and more.

Macro	ADAPT/renderer	Full PG 2.17+	Notes
PGML.pl	Yes	Yes	Always load this for PGML blocks
PGstandard.pl	Yes	Yes	Core PG functions
MathObjects.pl	Yes	Yes	Required for Real(), Compute(), Formula()
niceTables.pl	Yes	Yes	The only supported way to create tables (see Making Tables with niceTables)

Context macros: Most available, some missing

Context macros: Most available, some missing reference.

Macro	ADAPT/renderer	Full PG 2.17+	Notes
contextUnits.pl	Yes	Yes	Unit-aware parsing for science problems
contextReaction.pl	Yes	Yes	Chemical reaction input validation
VectorListCheckers.pl	No	Yes	Vector list validation not available in subset

What this means for problem authors

- **Use PopUp and RadioButtons:** These work reliably across all PG versions and are available in the ADAPT/renderer subset.
- **Avoid CheckboxList:** It's not in the PG 2.17 subset. Use RadioButtons per statement instead (see [Randomized MC True and False Statements](#) for the pattern).
- **Check the allowlist before copying:** Online examples may use macros that aren't in the subset. Always verify against the allowlist in `docs/PG_2_17_RENDERER_MACROS.md`.
- **Use niceTables.pl for all tables:** HTML table tags (`<table>` , `<tr>` , `<td>`) are blocked. niceTables.pl is the only supported way to create tables.
- **Test locally:** Use the local renderer ([Testing and Debugging](#)) to catch missing macros before deploying to ADAPT.

When a macro is missing

If you try to load a macro that's not in the ADAPT/renderer subset, you'll see an error. Here's how to diagnose and fix it.

Symptoms of a missing macro

- **"Can't locate [MacroFile].pl" error:** The macro file doesn't exist in the renderer's macro search path.
- **"Undefined subroutine" error:** A function from a missing macro was called (e.g., `CheckboxList()` without `parserCheckboxList.pl`).
- **Widget doesn't appear:** The problem renders but the expected checkbox list, dropdown, or other widget is missing.

How to diagnose

1. **Check the allowlist:** Open `docs/PG_2_17_RENDERER_MACROS.md` and search for the macro file name. If it's not listed, it's not available.
2. **Test locally with renderer:** Run the problem through the local renderer ([Setting Up the PG Renderer](#)). The error message will tell you which macro failed to load and where it was called.
3. **Search for alternatives:** Look for similar functionality in macros that ARE in the allowlist. For example, use `PopUp` instead of `DropDown`.

Common workarounds

Common workarounds: missing macro, what it provided, and workaround.

Missing macro	What it provided	Workaround
<code>parserCheckboxList.pl</code>	Checkbox lists for multiple correct answers	Use RadioButtons per statement (True/False), see Randomized MC True and False Statements
<code>DropDown</code> (PG 2.18+)	Modern dropdown widget	Use <code>PopUp</code> from <code>parserPopUp.pl</code> (same functionality)
<code>VectorListCheckers.pl</code>	Vector list validation helpers	Use MathObjects Vector context directly or simplify to scalar answers

Prevention

- Before copying code from online examples, check the macro list against the allowlist.
- Use templates from [Minimal Templates](#), which are verified to work with the PG 2.17 subset.
- Test problems locally ([Setting Up the PG Renderer](#)) before deploying to ADAPT.

Science-native contexts and grading helpers

These macros extend PG's answer checking to handle science-specific input formats like units, chemical reactions, and scientific notation. Use them when the standard Numeric or String contexts are too restrictive for what you want students to enter.

Science-native contexts and grading helpers reference.

Use it when	Macro to load	Why it matters
Units are part of correctness	contextUnits.pl	Supports unit-aware parsing and validation.
You need chemical reaction input	contextReaction.pl	Validates reactants/products and reaction structure.
Multiple blanks depend on each other	parserMultiAnswer.pl	Ties related blanks to one checker.
Parts should contribute different weights	weightedGrader.pl	Supports weighted scoring across parts.
Order should not matter in a list	unorderedAnswer.pl	Grades set-style answers without order.
Answer must include physical units (e.g., "9.8 m/s^2")	parserNumberWithUnits.pl	Parses and validates a number paired with a unit string.
Answer is a formula with units (e.g., "2*pi*r m")	parserFormulaWithUnits.pl	Extends FormulaWithUnits to accept symbolic expressions with units.
Answer should be in scientific notation	contextScientificNotation.pl	Requires answers like 6.02E23 or 6.02 x 10^23. Useful for biology measurements.
Answer must be a fraction, not a decimal	contextFraction.pl	Requires fractional form; rejects decimal equivalents.
Answer is a percentage	contextPercent.pl	Parses answers like "45%" and validates the percent sign.
You need a custom answer checker	answerCustom.pl	Build a fully custom grading subroutine when standard checkers are insufficient.
Partial credit for multi-part problems	PGgraders.pl	Provides full_partial_grader and other grading strategies.

Tables, layouts, and visual tools

These macros control how content is displayed, from data tables to scaffolded multi-part prompts to generated graphs. Most biology problems need at most niceTables.pl for data presentation.

Tables, layouts, and visual tools reference.

Use it when	Macro to load	Why it matters
You need structured tables or aligned blanks	niceTables.pl / unionTables.pl	Keeps data displays clean and readable.
You want scaffolded, multi-part prompts	scaffold.pl	Supports structured sections without custom logic.
An interactive graph tool is required	parserGraphTool.pl	Provides interactive graph widgets.
You need a TikZ-generated image in the problem	PGtikz.pl	Renders TikZ code to an image at display time.
You need a LaTeX-generated diagram or figure	PGlateximage.pl	Renders arbitrary LaTeX snippets to images.

Symptoms when missing

Symptoms when missing: symptom, likely missing, and fix.

Symptom	Likely missing	Fix
PGML renders as raw text	PGML.pl	Load PGML.pl before BEGIN_PGML.
Radio buttons do not appear	parserRadioButtons.pl	Add parserRadioButtons.pl to loadMacros().
Checkbox list does not appear	parserCheckboxList.pl (not available)	Use per-statement RadioButtons or matching instead.
MathObjects are undefined	MathObjects.pl	Load MathObjects.pl before using Real/Compute.

Local render shows the line and missing macro when ADAPT only reports a generic error ([Testing and Debugging](#)).

Graph, plotting, and image macros

The renderer includes several macros for generating figures and plots. Most biology problems do not need dynamic graphs, but these are available when a problem requires a visual.

Graph, plotting, and image macros: macro, what it does, and when to use it.

Macro	What it does	When to use it
PGgraphmacros.pl	Creates 2D function plots using the WWPlot/Fun system.	Simple function graphs (growth curves, dose-response). See Making Graphs .
PGtikz.pl	Renders TikZ code to an image.	Custom diagrams, flowcharts, or annotated figures.
PGlateximage.pl	Renders arbitrary LaTeX to an image.	Complex LaTeX-only figures or chemical structures.
parserGraphTool.pl	Interactive graph widget where students plot points or draw lines.	Problems that ask students to construct a graph.
LiveGraphics3D.pl	Embeds interactive 3D graphics.	Rarely needed for life-science problems. Available but niche.
PGstatisticsGraphMacros.pl	Pre-built statistical graph types (histograms, scatter plots).	Statistics-oriented problems with data visualization.

Statistics and numerical macros

These macros are available in the renderer and are useful for quantitative biology, biostatistics, or ecology problems that involve data analysis.

Statistics and numerical macros: macro, what it provides, and notes.

Macro	What it provides	Notes
PGstatisticsmacros.pl	Statistical functions: mean, standard deviation, normal and t-distribution values.	Useful for biostatistics problems involving hypothesis testing or confidence intervals.
PGnumericalmacros.pl	Numerical methods: interpolation, numerical integration, Newton's method.	Rarely needed for biology. More common in engineering or physics.
RserveClient.pl	Connects to an R statistical computing server from within PG.	Requires a running Rserve instance. Not available in most ADAPT deployments.

Matrix and linear algebra macros

Matrix macros are available in the renderer but are rarely needed for biology or life-science problems. They appear frequently in math and engineering problem libraries.

Matrix and linear algebra macros: macro, what it provides, and notes.

Macro	What it provides	Notes
PGmatrixmacros.pl	Basic matrix display and input helpers.	Use for problems involving transition matrices (population genetics, Markov chains).
MatrixCheckers.pl	Answer checkers for matrix-valued answers.	Validates student matrix input against correct answer.
MatrixReduce.pl	Row reduction and RREF utilities.	Rarely needed for biology.
contextMatrixExtras.pl	Extra MathObject operations for matrices.	Extends the Matrix context with additional operations.

Complex number and specialized math macros

These macros exist in the renderer for completeness but are almost never needed in biology or life-science problems. They are listed here so you recognize them if encountered in borrowed code.

Complex number and specialized math macros reference.

Macro	What it provides	Relevance to biology
PGcomplexmacros.pl	Complex number arithmetic and display.	None. Remove if found in copied code.
contextComplexExtras.pl / contextComplexJ.pl	Extended complex number contexts (j instead of i).	None. Engineering-oriented.
PGdiffeqmacros.pl	Differential equation display and solution helpers.	Rare. Could appear in population dynamics or pharmacokinetics problems.
PGpolynomialmacros.pl	Polynomial manipulation utilities.	None for typical biology problems.
LinearProgramming.pl	Simplex method and LP helpers.	None. Operations research only.

Quick reference: which macro for which task

Use this table when you know what you want the problem to do but are not sure which macro provides it.

Quick reference: which macro for which task: i want to... and load this macro.

I want to...	Load this macro
Ask a multiple-choice question	parserRadioButtons.pl
Ask a dropdown selection question	parserPopUp.pl
Ask for a numeric answer with units	parserNumberWithUnits.pl
Require scientific notation in the answer	contextScientificNotation.pl
Accept a chemical reaction as input	contextReaction.pl
Require a fraction (no decimals)	contextFraction.pl
Accept a percentage answer like "45%"	contextPercent.pl
Display a data table in the problem	niceTables.pl
Create a multi-part scaffolded problem	scaffold.pl
Show a dynamically generated graph or plot	PGgraphmacros.pl (or PGtikz.pl for custom diagrams)
Let students arrange steps in order	draggableProof.pl
Give partial credit by part	PGgraders.pl (or weightedGrader.pl for weighted scoring)
Grade related answer blanks together	parserMultiAnswer.pl
Write a custom answer checker from scratch	answerCustom.pl
Provide targeted feedback for common mistakes	answerHints.pl
Ask for a free-response essay	PGessaymacros.pl

Do not cargo-cult

- Remove macros you do not use and re-render.
- Load macros only once per file.
- Let dependencies load themselves; do not list everything "just in case."

Next, depending on what you are doing:

- Not sure where the macro block belongs? Go to [Sections within a PG question](#).
- Need a clean starter .pg skeleton? Go to [PG Problem Files](#).
- Copied legacy code? Go to [Legacy PG and deprecated patterns](#).

This page titled [2.5: Common PG Macros](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.6: Legacy PG and deprecated patterns

This page helps you recognize legacy PG patterns that show up in older problem libraries and online examples. The goal is to translate the useful idea into a PGML-first structure (see [Introduction to PGML](#) for modern PGML) without carrying forward fragile formatting or scattered grading logic.

If you pasted code from an old problem and it uses `BEGIN_TEXT`, `&ANS`, or raw Perl formatting, start here.

If you see a PopUp example online, treat it as a historical artifact, not a recommendation.

Use this page when: a copied problem looks strange or grades unpredictably.

Copy and edit (PGML-first rewrite)

```
# Legacy text block (do not use)
# BEGIN_TEXT
# The stock concentration is $c_stock uM.$PAR
# Find the final concentration: \{ans_rule(10)\} uM
# END_TEXT

# PGML-first rewrite
BEGIN_PGML
The stock concentration is [c_stock] μM.

Find the final concentration: [_____]{c_final} μM
END_PGML
```

Common failure and fix

```
Broken: $PAR and BEGIN_TEXT blocks mixed with PGML
What you will see: raw $PAR output or missing blanks
Fix: move all prompt text into BEGIN_PGML/END_PGML
```

Rewrite recipe (fast path)

1. Identify the story text and move it into a PGML block.
2. Move all numbers, ranges, and answer objects into Setup.
3. Attach each answer object directly to the blank in PGML.
4. Remove unused macros and leftover text-formatting codes.
5. Render a few seeds to confirm the prompt, units, and grading.

Legacy patterns to recognize (and replace)

Legacy patterns to recognize (and replace) reference.

Legacy pattern	Why you see it	PGML-first replacement
<code>beginproblem()</code> call in DOCUMENT section	Found in 78% of OPL files. Was once required for initialization but is now handled by <code>DOCUMENT()</code> .	Delete the line. <code>DOCUMENT()</code> alone is sufficient.
BEGIN_TEXT / END_TEXT blocks	Pre-PGML format for the prompt text.	Use BEGIN_PGML / END_PGML for readable prompt text.
\$PAR, \$BR, and manual line breaks	Older formatting controls inside text blocks.	Use normal paragraphs, lists, or tables in PGML.
String-built HTML (" ". \$x . " ")	Manual HTML assembly in PG strings.	Use PGML with variable inserts for clean text.
<code>ANS(num_cmp(...))</code> or <code>ANS(str_cmp(...))</code>	<code>num_cmp</code> appears in 46% and <code>str_cmp</code> in 7% of OPL files. Legacy evaluators from the pre-MathObjects era.	Use MathObjects and PGML inline blanks: <code>[_]{ \$answer }</code> . See the table below for specific replacements.
<code>ans_rule()</code> without nearby answer object	Older blank creation with separate answer logic.	Attach answer objects directly to PGML blanks.
Manual rounding inside strings	Legacy formatting for display-only answers.	Use MathObjects and explicit tolerance rules.

Deprecated or fragile behaviors

Deprecated or fragile behaviors reference.

Pattern	Why it is fragile	Preferred approach
<code>NchooseK(\$N, \$K)</code>	Deprecated in PGchoicemacros.pl. Very common in OPL (1,000+ files). Returns K indices chosen from 0..N-1.	Use <code>random_subset(\$K, 0..\$N-1)</code> from PGstandard.pl, or build an index array and <code>@list[@indices]</code> .
<code>shuffle(\$N)</code>	Deprecated in PGchoicemacros.pl. Returns a random permutation of indices 0..N-1.	Use <code>random_subset(\$N, 0..\$N-1)</code> for a full permutation, or Perl's built-in list operations.
<code>new_match_list</code> with <code>print_q / print_a</code>	PGchoicemacros.pl legacy match list generates its own form fields, requiring <code>ANS(str_cmp(...))</code> .	Use PopUp widgets (<code>parserPopUp.pl</code>) with PGML inline blanks. See Macro Demonstrations for both patterns.
PopUp menus for choices	Legacy UI pattern and inconsistent rendering.	Use DropDown or radio/checkbox patterns.
Hidden grading logic in text assembly	Hard to debug and easy to mismatch with the prompt.	Compute answers in Setup and attach to blanks.
Multiple unrelated Context() changes	Inconsistent parsing rules across blanks.	Set Context once and override only if necessary.
<code>num_cmp()</code> , <code>str_cmp()</code> , <code>fun_cmp()</code>	Legacy evaluators from pre-MathObjects era. <code>num_cmp</code> alone appears in 46% of OPL files.	Use MathObjects (<code>Compute()</code> , <code>String()</code> , <code>Formula()</code>) with <code>->cmp()</code> or PGML blanks. See table below.
<code>TEXT(beginproblem())</code>	No longer needed. Appears in 78% of OPL files but <code>DOCUMENT()</code> handles initialization.	Delete the line entirely. Use <code>DOCUMENT();</code> alone.
<code>AnswerFormatHelp.pl</code>	Entire macro file deprecated in PG 2.17.	Use <code>helpLink()</code> from PGbasicmacros.pl (loaded via PGstandard.pl).
<code>compoundProblem.pl</code> / <code>compoundProblem5.pl</code>	Deprecated in PG 2.19. Legacy multi-section problem macros.	Use <code>scaffold.pl</code> with <code>Scaffold::Begin()</code> and <code>Section::Begin()</code> .
Old macro families loaded "just in case"	Adds noise and can mask missing dependencies.	Load only the macros needed for the prompt.

Legacy answer evaluators (very common in copied code)

Before MathObjects existed, every problem used standalone evaluator functions like `num_cmp()` and `str_cmp()`. These still work but are discouraged because MathObjects provide richer error messages, tolerance handling, and direct attachment to PGML blanks. If you copy a problem from the OPL, there is roughly a 50% chance it uses at least one of these legacy evaluators.

Copy and edit (answer evaluator rewrite)

```
# Legacy pattern (do not use in new problems)
# $answer = 42;
# ANS(num_cmp($answer));
# ANS(num_cmp($answer, tol => 0.01));

# MathObjects + PGML rewrite
$answer = Compute("42");
BEGIN_PGML
```

```
Enter the answer: [_____]{$answer}
END_PGML
# Tolerance is set via: $answer->cmp(tolerance => 0.01)
```

Copy and edit (answer evaluator rewrite) reference.

Legacy evaluator	OPL prevalence	What it does	MathObjects replacement
<code>num_cmp(\$x)</code>	46% (33,804 files)	Compares student answer to a number. Supports tolerance, strings like "DNE".	<code>\$x = Compute("42"); then PGML [_]{\$x} or ANS(\$x->cmp())</code>
<code>str_cmp("text")</code>	7% (4,958 files)	Compares student answer to a string. Used for word answers, matching.	<code>\$x = String("text"); then PGML [_]{\$x} or ANS(\$x->cmp())</code>
<code>fun_cmp("x^2")</code>	Common in math OPL	Compares student formula by sampling points.	<code>\$f = Formula("x^2"); then PGML [_]{\$f} or ANS(\$f->cmp())</code>
<code>std_num_str_cmp()</code>	Low	Hybrid number/string comparator. Formally deprecated in PG 2.17.	Use <code>Compute()</code> with appropriate Context.
<code>numerical_compare_with_units()</code>	Low	Number comparison with unit checking. Formally deprecated in PG 2.17.	Use <code>NumberWithUnits()</code> from <code>parserNumberWithUnits.pl</code> .

Key point: `num_cmp()` and `str_cmp()` are not formally marked deprecated in the PG source (they have been rewritten to use MathObjects internally), but the PG documentation recommends using MathObjects `->cmp()` methods directly. New problems should always use MathObjects.

[beginproblem\(\): safe to delete](#)

`beginproblem()` appears in 78% of OPL files (56,953 of 72,734). It was once required to initialize problem output, but `DOCUMENT()` now handles all initialization. If you copy a problem that includes `TEXT(beginproblem())`, simply delete that line.

```
# Legacy (found in most OPL files)
# DOCUMENT();
# TEXT(beginproblem());

# Modern (just DOCUMENT is enough)
DOCUMENT();
```

[PGchoicemacros.pl: widely used, mostly deprecated](#)

`PGchoicemacros.pl` appears in over 1,000 OPL files, but most of its utility functions are now deprecated. If you copy a problem that loads `PGchoicemacros.pl`, check which functions it actually uses and consider whether a modern replacement exists.

[PGchoicemacros.pl: widely used, mostly deprecated reference.](#)

Deprecated function	Replacement	Example rewrite
<code>NchooseK(5, 3)</code>	<code>random_subset()</code>	<code>@pick = random_subset(3, 0..4);</code>
<code>shuffle(5)</code>	<code>random_subset()</code>	<code>@perm = random_subset(5, 0..4);</code>
<code>new_match_list</code>	PopUp widgets	Build PopUp objects and use PGML inline blanks (see Matching).
<code>new_select_list</code>	RadioButtons	Use <code>parserRadioButtons.pl</code> with PGML (see Multiple Choice).
<code>new_multiple_choice</code>	RadioButtons	Use <code>parserRadioButtons.pl</code> with PGML (see Multiple Choice).

Not deprecated: `PGsort` is sometimes confused with `PGchoicemacros.pl` but is actually a core function from `PGbasicmacros.pl` (loaded via `PGstandard.pl`). It does not require loading `PGchoicemacros.pl` and is not deprecated.

Apply it today

- When you copy an older example, rewrite the text block into PGML first.
- Keep the macro list short and focused on what the prompt actually uses.
- Use seeds to reproduce any unexpected grading or formatting behavior (see [Common Mistakes and How to Fix Them](#) for common mistakes).

Next, depending on what you are doing:

- Once it is PGML-first, go back to [PG Problem Files](#) and rebuild from the minimal skeleton.
- Need macro help? Go to [Common PG Macros](#).
- Not sure where setup ends and PGML begins? Go to [Sections within a PG question](#).

This page titled [2.6: Legacy PG and deprecated patterns](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

2.7: Future PG version features

This guide targets the PG 2.17 subset available in ADAPT and the standalone `webwork-pg-renderer`. Newer PG releases add features that are not yet available in this environment, but understanding what is coming helps you plan problems that will benefit from future upgrades and avoid reinventing patterns that already exist upstream.

Use this page when: you see a feature in online examples or the OpenWeBWork wiki that does not work in ADAPT and want to know which PG version introduced it.

How to read this page

- Features listed under each version are *cumulative*: PG 2.19 includes everything from 2.18 and earlier.
- Items marked **(PG only)** are changes to the problem-authoring language. Items marked **(WW only)** are server or UI changes that do not affect .pg files.
- The textbook currently targets **PG 2.17**. Everything below that line already works in ADAPT.

PG 2.17 (current ADAPT baseline)

This is what ADAPT and the standalone renderer support today. Key features already documented in this textbook:

- Java and Flash applet support removed (problems using them will show a warning).
- GraphTool additions: point tool, three-point quadratic tool, four-point cubic tool, and keyboard controls for accessibility.
- DraggableProof updated to work in Gateway quizzes when written correctly.
- MathQuill input improvements for expressions and units.

PG 2.18 features (not yet in ADAPT)

PG 2.18 is the first version with significant new authoring capabilities beyond what ADAPT currently supports.

New PGML syntax

PG 2.18 is the first version that reduces the need for many of the workarounds you see in this textbook, especially for inserting images into PGML and creating checkbox (select-all-that-apply) questions.

New PGML syntax: feature and what it does.

Feature	What it does
PGML images	Insert images with <code>[!alt text!]{image source}{width}{height}</code> directly in PGML instead of using command substitution or HTML.

New macros

New macros: macro and description.

Macro	Description
<code>plotly3D.pl</code>	Plot 3D curves and surfaces using the plotly.js library. Useful for visualizing molecular structures, reaction coordinate diagrams, or any data that needs a third axis.
<code>randomPerson.pl</code>	Generate a random person name with correct pronoun methods (he/she/they). Produces inclusive, readable story problems.
<code>parserCheckboxList.pl</code>	Native checkbox answer type for select-all-that-apply questions. Replaces the RadioButtons-per-statement workaround documented in Multiple Answer .
<code>parserRadioMultiAnswer.pl</code>	Tie a radio answer together with dependent answer blanks that change based on the radio choice. Useful for branching questions.
<code>parserLinearRelation.pl</code>	Context for lines and linear inequalities as MathObjects.
<code>specialTrigValues.pl</code>	Standard trig values on the unit circle as a helper macro.

Updated macros

- **niceTables.pl**: Completely overhauled. If you are writing table-heavy problems, the 2.18 version has better options and cleaner syntax than what is documented in [Making Tables with niceTables](#).
- **parserCheckboxList.pl** / **parserRadioButtons.pl**: Past-answer display now shows meaningful labels instead of B0, B1, etc.
- **parserGraphTool.pl**: Number line mode for intervals, default answer checkers, a `generateAnswerGraph` method for solutions, and a dynamic help system.
- **parserPopUp.pl**: New `DropDown` method as a modern replacement for `PopUp`.

Other PG 2.18 changes

- MathQuill: typing `deg` produces a degree symbol; trig and log functions auto-wrap arguments in parentheses.
- Units: students can enter $^{\circ}\text{C}$ and $^{\circ}\text{F}$ directly; many new units added (angstroms, microseconds, picometers, etc.); plural forms accepted (30 ft, 30 feet, 1 foot).
- Hardcopy generation revamped with a theme system (eleven themes included).
- LTI 1.3 authentication module added.

PG 2.19 features (not yet in ADAPT)

PG 2.19 adds several quality-of-life improvements for PGML authoring.

New PGML syntax

PG 2.19's biggest authoring win is native PGML tables, which means you can build tables directly inside a PGML block without the explicit `loadMacros("niceTables.pl")` call you currently need.

New PGML syntax: feature and what it does.

Feature	What it does
PGML tables	Create tables directly in PGML blocks using <code>niceTables.pl</code> syntax. The macro is automatically loaded when using PGML, so no explicit <code>loadMacros</code> call is needed.
HTML div/span tags in PGML	Add HTML <code><div></code> or <code></code> tags within PGML using <code>[< ... >]</code> syntax. This replaces the MODES workaround for wrapper divs documented in Randomized Matching Problems .
Horizontal rules fixed	PGML horizontal rules updated and made reliable.

New macros and MathObjects

New macros and MathObjects: macro and description.

Macro	Description
<code>contextBoolean.pl</code>	Boolean MathObject context with parsing of boolean expressions, reduction rules with AND/OR, and customizable true/false symbols including unicode. Relevant for logic gates in genetics or Boolean search problems.
<code>parserMultipleChoice.pl</code>	Umbrella macro that loads all multiple choice parsers at once: <code>PopUp</code> , <code>CheckboxList</code> , <code>RadioButtons</code> , <code>RadioMultiAnswer</code> . Simplifies the <code>loadMacros</code> block.

Other PG 2.19 changes

- **Matrix MathObjects**: New methods for creating zero, permutation, and elementary matrices; tests for triangular, orthogonal, diagonal, symmetric, row echelon, and reduced row echelon forms.
- **Fraction MathObjects**: New `num` and `den` methods for extracting numerator and denominator.
- **Compute behavior**: Calling `Compute` on a number now creates a Real directly instead of stringifying and passing to Formula.
- **Knowls**: Reworked to open in modal dialogs instead of inline expansion.

- The attempts table has been removed; feedback is now associated directly with each answer blank. **(WW only)**
- The PG editor now has an option to convert problems to PGML (use with care, conversion is not complete). **(WW only)**

Breaking changes in PG 2.19

- **parserPopUp.pl default change:** When the correct answer is a non-negative integer, it is now treated as an index into the answer array (consistent with other macros) rather than the literal answer string. Problems using integer values as PopUp answers may need updating.

Deprecated in PG 2.19

- **compoundProblem.pl** (all versions: `compoundProblem.pl`, `compoundProblem2.pl`, `compoundProblem5.pl`): Migrate to `scaffold.pl` instead.

PG 2.20 features (current release)

PG 2.20 focuses on graphing tool improvements, modern plotting, units as MathObjects, and cleanup of deprecated patterns.

New macros and features

PG 2.20 brings unit-aware MathObjects through `contextUnits.pl`, so you can grade answers that include units as part of the MathObject system instead of relying on external workarounds. It also introduces a modern plotting macro and a new context extension mechanism.

New macros and features: macro / feature and description.

Macro / Feature	Description
<code>plots.pl</code>	Modern, efficient plotting macro for a wide variety of graph types. Intended as a more capable replacement for older graphing approaches.
<code>contextUnits.pl</code>	Units implemented as MathObject classes. Students can enter answers with units and have them graded as part of the MathObject framework.
<code>contextExtensions.pl</code>	New context extension mechanism. Fraction contexts are now extensions of the parent context rather than standalone replacements.

Updated macros

- **parserGraphTool.pl:** Four new tools added: polygon tool, sine wave tool, vector tool, and improved fill tool.
- **contextReaction.pl:** Improvements to chemical reaction context. Relevant for chemistry and biochemistry problems.
- **parserPopUp.pl:** Full HTML now supported in drop-down menu items (bold, subscripts, etc.).
- **Degree n matrices:** Improvements to matrices/tensors with more than two dimensions.
- **Rserve interface:** Rewritten within PG; the `Statistics::R::IO::Rserve` package is no longer required.

Breaking changes in PG 2.20

- **custom_problem_grader_0_60_100 removed:** Problems must be converted to use `custom_problem_grader_fluid` instead. This is relevant if you copy older problems that use the 0/60/100 grading scale.
- **Degree 1 vs degree 2 matrix comparison:** Changed behavior when comparing a degree 1 matrix and a degree 2 matrix.

WeBWork 2.20 server changes (WW only)

- Problem editor upgraded to CodeMirror 6 with accordioning, autocompletion, and improved syntax highlighting.
- Security vulnerability fixes and database efficiency improvements.
- Problems can now be loaded directly from the Problem Editor or from sample problems.

What this means for ADAPT authors today

Until ADAPT upgrades its PG version, the workarounds documented in this textbook remain necessary. Here is a summary of what changes when ADAPT eventually updates:

The following table maps each workaround documented in this textbook to the PG version that eliminates it, so you can plan which problems to update first when ADAPT upgrades.

What this means for ADAPT authors today reference.

Current workaround	Available in	What replaces it
RadioButtons per statement for select-all-that-apply (Multiple Answer)	PG 2.18	Native <code>parserCheckBoxList.pl</code> with proper checkbox UI
PopUp for dropdown menus	PG 2.18	<code>DropDown</code> method in <code>parserPopUp.pl</code>
MODES wrapper for flexbox divs in matching layouts (Randomized Matching Problems)	PG 2.19	<code>[< ... >]</code> div/span syntax in PGML
Explicit <code>loadMacros("niceTables.pl")</code> for tables	PG 2.19	<code>niceTables.pl</code> auto-loaded with PGML
Command substitution or HTML for images in PGML	PG 2.18	Native <code>[!alt!]{src}{w}{h}</code> PGML image syntax
No 3D visualization available	PG 2.18	<code>plotly3D.pl</code> for 3D curves and surfaces
TikZ images not available	PG 2.16+	<code>PGtikz.pl</code> for LaTeX-quality vector graphics (requires server-side LaTeX; not available in ADAPT's renderer subset)
Plain text in PopUp/DropDown menu items	PG 2.20	Full HTML in drop-down items (bold, subscripts, special characters)
<code>custom_problem_grader_0_60_100</code>	PG 2.20	Removed. Use <code>custom_problem_grader_fluid</code> instead
<code>PGgraphmacros.pl</code> for static plots	PG 2.20	<code>plots.pl</code> for modern, efficient graph generation
No native unit-aware answer checking	PG 2.20	<code>contextUnits.pl</code> for units as MathObject classes

Apply it today

- Write problems using the PG 2.17 patterns in this textbook. They will continue to work in newer versions.
- When ADAPT upgrades, revisit select-all-that-apply problems (switch from RadioButtons fallback to CheckBoxList) and matching layouts (replace MODES wrappers with PGML div syntax).
- Do not use TikZ, plotly3D, plots.pl, contextUnits.pl, CheckBoxList, RadioMultiAnswer, or DropDown in problems intended for the current ADAPT environment.
- If you copy a problem that uses `custom_problem_grader_0_60_100`, replace it with `custom_problem_grader_fluid` (required in PG 2.20, works in all versions).

Next, depending on what you are doing:

- Need the current macro allowlist? Go to [Common PG Macros](#).
- Copied legacy code and it looks strange? Go to [Legacy PG and deprecated patterns](#).
- Ready to write your first problem? Go to [PG Problem Files](#).

This page titled [2.7: Future PG version features](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

3: PGML (PG Markup Language)

PGML (PG Markup Language) is the recommended way to write the student-facing text of a WeBWorK problem. It is designed to replace older authoring patterns that required long blocks of text and manual escaping.

Start here (task first)

- I need one blank that grades correctly -> [Answer blanks and answers](#) (Core).
- I need a multi-step lab workflow prompt -> [Lists and workflows](#) (Core).
- I need a small dataset table students read from -> [Tables and structured data](#) (Core).
- My prompt is too long and hard to scan -> [Layout controls](#) (Optional).
- I need to insert computed or helper-generated output -> [Command substitution and inserts](#) (Optional).

This chapter is the core of the book: most science-friendly question writing can be done by combining a small PG setup with a clear PGML prompt.

What PGML gives you

- Readable prompts that look similar to the rendered output.
- Math in TeX or calculator notation, with consistent delimiters.
- Answer blanks that can be tied directly to answers.
- Simple structure tools (headings, lists, tables) that fit science prompts.

PGML quick patterns

PGML quick patterns: task, use, and page.

Task	Use	Page
One blank that grades correctly	Answer blanks attached to answers	Answer blanks and answers
Multi-step lab workflow	Numbered lists	Lists and workflows
Small dataset table	DataTable	Tables and structured data
Long prompt cleanup	Headings and light emphasis	Layout controls
Computed or helper output	Command substitution	Command substitution and inserts
PGML basics and notation	Minimal PGML patterns	Introduction to PGML (Optional)

The subsections in this chapter cover the small set of PGML features you will use constantly: math notation, answer blanks, lists for workflows, tables for data, and a small "escape hatch" for advanced insertion when PGML alone is not enough.

Core pages: [Answer blanks and answers](#) (Answer blanks) and [Lists and workflows](#) (Lists). Optional pages: [Layout controls](#) (Layout) and [Command substitution and inserts](#) (Command substitution). Use [Introduction to PGML](#) as a quick primer if you want notation examples.

If you are unsure which PGML pattern to use, start with the simplest option that clearly communicates the task. Most grading issues in life science questions come from misaligned structure rather than missing features. When in doubt, clarity beats cleverness.

[3.1: Introduction to PGML](#)

[3.2: Answer blanks and answers](#)

[3.3: Lists and workflows](#)

[3.4: Tables and Structured Data](#)

[3.5: Layout controls](#)

[3.6: Command substitution and inserts](#)

This page titled [3: PGML \(PG Markup Language\)](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

3.1: Introduction to PGML

PGML is a markup language for writing the student-facing text of a WeBWorK problem. It is designed to replace older authoring styles that required long blocks of text and careful escaping.

In this book, [PG](#) is treated as minimal setup code, and PGML is where you spend most of your authoring time.

Use this when: you want the prompt to read like a worksheet while keeping the grading logic close to the text students see.

Copy and edit (minimum viable PGML)

This template shows a minimal PGML block with inline math, variable substitution, and one answer blank.

```
$km = Real(2.5);
$substrate = Real(12);
$rate = Real(0.48);

BEGIN_PGML
For this enzyme, [ $K_m = [\$km]$ ]  $\mu\text{M}$  and [ $[S] = [\$substrate]$ ]  $\mu\text{M}$ .
Report the rate [ $v$ ] in  $\mu\text{M/s}$ :

[ $v =$ ] [____]{$rate}
END_PGML
```

Customize the variable names, the prompt text, and the answer expression in braces.

Common failure and fix

Without backtick delimiters, subscripts and Greek letters render as plain text instead of formatted math.

```
Broken: Km = 2.5  $\mu\text{M}$ 
Fix: [ $K_m$ ] = 2.5  $\mu\text{M}$ 
```

What PGML is good at

- Readable prompts that look similar to the rendered output.
- Math written in TeX or calculator notation with consistent delimiters.
- Answer blanks tied directly to answer checkers.
- Structured formatting for science prompts (lists, tables, and short sections).

Most instructors do not need to learn PGML exhaustively. In practice, a small set of patterns covers the majority of biology and life science questions: inline math, answer blanks, short lists, and simple tables. This chapter focuses on those patterns so you can write readable prompts without treating PGML as another programming language.

Math notation (TeX and calculator notation)

PGML supports two styles for writing mathematics: TeX and calculator notation. Both styles support inline math and displayed math, and you should pick the one that matches the situation.

Use TeX when you care about typesetting (for example, [K_m] or [$V_{\max}$]). Use calculator notation when you care about readability and copy-paste (for example, [Ct] or [$C1 \cdot V1/V2$] for a dilution).

TeX math delimiters

- Inline: [$\dots$]
- Display-style inline: [$\displaystyle \dots$]

- Display (own line): `[`` ` ` ... `` ` `]`

Calculator notation delimiters

- Inline: `[ : ... : ]`
- Display-style inline: `[ :: ... :: ]`
- Display (own line): `[ ::: ... ::: ]`

Calculator notation delimiters: task, delimiter, and example.

Task	Delimiter	Example
Variable name	TeX inline	<code>[`K_m`]</code>
Short equation	Calculator inline	<code>[: C1*V1/V2 :]</code>
Long equation	TeX display	<code>[`` ` ` V = V_{max} * S / (K_m + S) `` ` `]</code>

Science-friendly practice

- Use inline math for variable names and short expressions.
- Use display math for longer equations or multi-step expressions.
- Keep units outside the math expression unless you are explicitly typesetting them.

Math formatting vs variable substitution

PGML uses square brackets for two different purposes. Confusing them is a common mistake.

Math formatting vs variable substitution: purpose, syntax, example, and more.

Purpose	Syntax	Example	Result
Math formatting	<code>[` ... `]</code>	<code>[`K_m`]</code>	Renders K with subscript m
Variable substitution	<code>[\$variable]</code>	<code>[\$km]</code>	Displays value: 2.5
Combined	<code>[`K_m = [ \$km ] `]</code>	<code>[`K_m = [ \$km ] `]</code>	Math with value inserted

Key rule: Use backticks when you want LaTeX math rendering. Use plain brackets (no backticks) when you want to display a variable's value as text.

Common mistake

Wrong: The answer is `[ `$radio->correct_ans` ]`
 Right: The answer is `[ $radio->correct_ans ]`

The wrong version tries to render "Blue" as LaTeX math. The right version displays the text value.

PGML is single-pass

PGML parses your text exactly once. This means that if you build PGML tag-wrapper syntax inside a Perl variable, PGML will not re-parse it. The variable's contents are inserted as literal text, so patterns like `[ <...> { [ ... ] } ]` stored in a string will not produce the expected formatting.

A related pitfall involves the `[ <` character sequence. If a Perl string assigned earlier in your code contains `[ <` , it can collide with PGML's own bracket-angle syntax and produce parser errors or garbled output. Watch for this when building HTML snippets or conditional text in Perl and then inserting them with `[ $var ]`.

The single-pass rule also means that `[ $... ]` PGML interpolation inside a `[ @ ... @ ]` eval block will not work. The eval block runs Perl; it does not re-enter the PGML parser. If you need formatted output from an eval block, return a string and let PGML display it.

If you are writing your first problem, go to [Answer blanks and answers](#) now.

Next, depending on what you are doing:

- Need a blank that grades correctly? See [Answer blanks and answers](#).
- Building a multi-step workflow prompt? See [Lists and workflows](#).
- Need a data table students read from? See [Tables and structured data](#).

This page titled [3.1: Introduction to PGML](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

3.2: Answer blanks and answers

Answer blanks are the center of most WeBWorK problems. In PGML, blanks are written as `[_____]` and the correct answer can be attached immediately after the blank using braces. For the full catalog of question types that use answer blanks, see [Different Question Types](#).

Use this when: you want the grading rule to be obvious and close to the prompt text students read.

Copy and edit

The following template shows a numeric answer blank with the correct value attached inline, so the grading rule lives right next to the prompt text.

```
$value = Real(12);
$answer = Real(24);

BEGIN_PGML
You measure a stock with [c = [$value]] mM.
What is twice the concentration?

Report your answer in mM:
[2c = ] [_____]{$answer} mM
END_PGML
```

To adapt this for your own problem, change the variable names, the PGML prompt text, and the answer expression inside the braces.

This error happens when you pass a numeric MathObject (like a Real) where PG expects a string answer. PG requires the type inside the braces to match the active Context, so passing `$ans2` (a Real) in a spot that needs a string produces a type mismatch.

Common failure and fix

```
Broken: [ATP = ] [_____]{$ans2}
Fix: [ATP = ] [_____]{"ATP"}
```

Many examples online define answers far away from the text that students see. That pattern works for short math problems but becomes fragile in biology-style prompts with multiple parts and narrative flow. Attaching each answer directly to its blank keeps the grading logic aligned with the story and makes later edits much safer.

Text answers in biology

Text answers are common in biology (gene/protein names, labels, or categories). Keep the accepted strings narrow and document what counts as correct to avoid over-accepting.

Free-text answers are appropriate when the response is a specific term—a gene name, molecule, or category label—rather than a number or formula. Use free-text instead of multiple choice when you want students to recall the term rather than recognize it from a list. Keep the accepted string set narrow to avoid grading ambiguity.

```
Context("String");
Context()->strings->add(
    ATP => {},
    atp => { alias => "ATP" },
    adenosine_triphosphate => { alias => "ATP" }
```

```
);
```

```
BEGIN_PGML
```

```
The molecule is [_____]{"ATP"}.
```

```
END_PGML
```

Expression and formula blanks

If the correct answer is a formula, use a quoted string or a Formula MathObject. Use `Compute()` or `Formula()` when the correct answer is an expression students type, not just a number.

```
$f = Compute("x^2-1");
```

```
BEGIN_PGML
```

```
[`y = `] [_____]{$f}
```

```
END_PGML
```

Why attach answers to blanks

When you attach an answer directly to a blank with `{$ans}`, PG registers the answer checker at parse time and binds it to that specific blank, so there is no separate answer-matching step that could go wrong. This matters most in multi-part problems where answer order can drift if answers and blanks are defined separately.

- The grading rule stays next to the text that asks for it.
- Multi-part science prompts are easier to maintain.
- It reduces the chance of mismatching answer order and prompt order. For common answer blank mistakes and how to fix them, see [Common Mistakes and How to Fix Them](#).

Next, depending on what you are doing:

- Writing a multi-step lab workflow? Go to [Lists and workflows](#).
- Need a data table students read from? Go to [Tables and structured data](#).
- If formatting is the only issue, try [Layout controls](#).

This page titled [3.2: Answer blanks and answers](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

3.3: Lists and workflows

Lists are one of the most useful PGML features for life science prompts because many tasks are multi-step workflows. Use lists to make expectations scannable and to reduce "I missed that part" errors. For a complete ordered-list question type, see [Ordered List](#).

Use this when: you want a lab-style sequence that doubles as a grading rubric.

Copy and edit

```
BEGIN_PGML
You are preparing a PCR reaction. Follow these steps:

1. Compute the final primer concentration in  $\mu\text{M}$ : [____]{$c_final}
2. Report the  $\text{MgCl}_2$  concentration in  $\text{mM}$ : [____]{$mg_final}
3. State whether the buffer is 1x or not: [____]{$buffer_ok}

Use the units shown in each step.
Round to two decimals.
END_PGML
```

Common failure and fix

```
Broken:
1. First step
Second paragraph without indent.
Fix:
    Second paragraph with four-space indent.
```

In lab-style questions, lists are not just formatting. They communicate order, expectations, and reporting structure to students. When you later revise a problem, changing list order often requires checking that the associated answer blanks still match the intended step.

Numbered lists

```
BEGIN_PGML
1. Compute the concentration.
2. Report the value with units.
END_PGML
```

Start a numbered list item with a number, a period, and a space. PGML will renumber items in the rendered output, so the specific numbers you type are not the important part.

To include a second paragraph within the same list item, indent the paragraph with four spaces.

```
BEGIN_PGML
1. First step with a paragraph.

    This stays inside the first item.
2. Second step
END_PGML
```

Bullet lists

```
BEGIN_PGML
* Use the values from the table.
* Keep units consistent.
END_PGML
```

Bulleted list items typically begin with an asterisk (`*`) or a dash (`-`).

Biology workflow example

```
BEGIN_PGML
You are preparing a PCR reaction mix. Follow these steps:

1. Compute the final primer concentration in  $\mu\text{M}$ .
2. Compute the final  $\text{MgCl}_2$  concentration in  $\text{mM}$ .
3. Decide whether the final buffer is 1x or not.
END_PGML
```

Watch out for auto-parsed list labels

PGML's list parser is aggressive. Any line that starts with a letter followed by a period (for example, `A.` or `B.`) can be interpreted as an ordered list item, even when you intended it as a plain label. This is especially common when writing matching-style questions where you use letters as answer choices.

Two easy fixes:

- Make the label bold: `*A.*` — the bold wrapper prevents list parsing.
- Escape the period: `A\.` — the backslash tells PGML that the period is literal text.

```
BEGIN_PGML
Match each description to the correct answer.

*A.* Mitochondria
*B.* Chloroplast
*C.* Ribosome
END_PGML
```

Authoring notes

- Keep each list item to one task.
- When a task requires a unit or a format, state it inside the list item.
- Use lists to make constraints visible (for example, "round to two decimals").

Next, depending on what you are doing:

- Need a single blank that grades correctly? Go to [Answer blanks and answers](#).
- Need a data table students read from? Go to [Tables and structured data](#).
- If grading is failing, go back to [Answer blanks and answers](#).

This page titled [3.3: Lists and workflows](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

3.4: Tables and Structured Data

Science prompts often require structured data. Prefer table helpers that produce accessible tables in HTML output, and avoid using tables purely for layout.

Use this when: students need to read values from a dataset before answering.

Copy and edit

Required macro: `niceTables.pl` must be loaded in the preamble to use `DataTable` or `LayoutTable`. For advanced table formatting with `niceTables`, see [Making Tables with niceTables](#).

```
# In preamble:
loadMacros("PGstandard.pl", "PGML.pl", "niceTables.pl", "PGcourse.pl");

# In setup:
$ct_treat = Real(21.2);

BEGIN_PGML
[@ DataTable([
    [ "Sample", "Ct (cycles)" ],
    [ "Control", 18.2 ],
    [ "Treatment", 21.2 ],
], caption => "qPCR results") @]*

The Ct for Treatment is [____]{$ct_treat}.
END_PGML
```

Common failure and fix

```
Broken: DataTable(...) without niceTables.pl loaded
Fix: add "niceTables.pl" to loadMacros()

Broken: LayoutTable(["Sample","Ct"],["Control",18.2]) for real data
Fix: DataTable(["Sample","Ct"],["Control",18.2])
```

Two table roles

- Data tables: real measurements or observations.
- Layout tables: spacing-only, when there is no meaningful row/column relationship.

Recommended helpers

- `DataTable(...)` : use when the table communicates data (measurements, observations, counts).
- `LayoutTable(...)` : use when the table is only for arranging content and there is no meaningful row/column relationship.

Whenever a table represents measurements, observations, or experimental results, treat it as data rather than layout. Using a data-oriented table makes the prompt clearer to students and reduces the chance that later formatting changes will alter meaning or grading behavior.

Tables should carry units in the header row, not scattered in every cell (for example "Ct (cycles)" or "Concentration (μM)").

Basic calling pattern

```
DataTable(  
  [  
    [ "Sample", "Ct" ],  
    [ "Control", 18.2 ],  
    [ "Treatment", 21.2 ],  
  ],  
  caption => "qPCR results",  
);
```

Authoring notes

- Use a data table whenever you want students to read values and compute from them.
- Keep tables small and label columns clearly; avoid wide tables with many decimals.
- Do not use a data table to position unrelated content; use `LayoutTable` for layout.

Next, depending on what you are doing:

- Need a multi-step workflow prompt? Go to [Lists and workflows](#).
- Need computed inserts or helper output? Go to [Command substitution and inserts](#).
- Need a single blank that grades correctly? Go to [Answer blanks and answers](#).

This page titled [3.4: Tables and Structured Data](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

3.5: Layout controls

Layout controls can improve readability for longer prompts, but fragile formatting is one of the fastest ways to create problems that render differently across systems. Prefer simple structure (headings, lists, and tables) and treat layout controls as an optional finishing touch.

Use **this when**: a prompt needs clearer scanning after the grading is already stable.

Copy and edit (minimal structure recipe)

```
BEGIN_PGML
# Data
Measured values are shown above.

## Tasks
1. Compute the rate in  $s^{-1}$ .
2. Report the final value with units.

### Report
[`Rate = `] [____]{$rate}  $s^{-1}$ 
END_PGML
```

Common failure and fix

```
Broken: *Measure* *_everything_* *carefully* and *report* *everything*.
Fix: Report the value in *mM* to *two decimals*.
```

Headings (structure)

Headings help students scan a multi-part prompt. In PGML, headings are written with # characters at the start of a line.

```
BEGIN_PGML
# Data
## Tasks
### Report
END_PGML
```

Emphasis (bold and italic)

- Bold: ****bold****
- Italic: *_italic_*
- Bold italic: ****_bold italic_****

Use emphasis to highlight the task being asked and the reporting expectations (units, rounding), not to decorate the story. For coloring and advanced emphasis options, see [Text Coloring and Emphasis](#).

Horizontal rules (sparingly)

Horizontal rules can separate major blocks, but headings are usually clearer.

```
----
```

Justification and centering (rare)

PGML supports simple right-justification and centering syntax. These are occasionally useful for a short "report" block, but they can be fragile and are easy to overuse.

```
BEGIN_PGML
>> Right-justified line

>> centered <<
END_PGML
```

Layout controls are best used sparingly. If a formatting change affects how answers are entered, checked, or interpreted, it is usually a sign that the structure should be simplified rather than refined. Clear structure matters more than visual polish.

Raw HTML and variable output

When a Perl variable contains HTML tags (such as `<span>` or `<div>`), PGML escapes them by default. To preserve the HTML, use the raw pass-through marker: `[$var]*`. Without the trailing asterisk, the student sees literal angle brackets instead of styled output. This is the same asterisk used after `[@ ... @]*` eval blocks.

Next, depending on what you are doing:

- If grading is failing, go back to [Answer blanks and answers](#).
- Need a data table students read from? Go to [Tables and structured data](#).
- Need computed inserts? Go to [Command substitution and inserts](#).

This page titled [3.5: Layout controls](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

3.6: Command substitution and inserts

Command substitution lets you run a small Perl expression and insert its output into the PGML prompt. This is useful for inserting images, computed strings, or helper-generated markup. Variables used in substitution are typically defined in the [Setup block](#).

Command substitution is powerful, but it should not be the first solution to a formatting or clarity problem. In most cases, PGML lists, tables, or explicit text are easier to read and easier to maintain. Use substitution only when the content genuinely depends on computed or conditional output.

Use this when: you need to insert computed output or helper-generated markup that PGML cannot express directly.

Copy and edit

```
$Ct = Real(21.2);
$rows = [
    [ "Sample", "Ct (cycles)" ],
    [ "Control", 18.2 ],
    [ "Treatment", 21.2 ],
];

BEGIN_PGML
[@ DataTable($rows, caption => "qPCR results") @]*
The Ct for Treatment is [____]{$Ct}.
END_PGML
```

Common failure and fix

```
Broken: [@ DataTable($rows) @]
Fix:   [@ DataTable($rows) @]*
```

Syntax

```
[@ some_perl_expression() @]
```

If the substitution produces output that is already formatted as PG/HTML, follow it with a star:

```
[@ some_perl_expression() @]*
```

Syntax: use it for and avoid it for.

Use it for	Avoid it for
Computed intermediate values or helper-generated tables.	Basic formatting that PGML already supports.
Reusable PG helpers that return formatted output.	Simple text that can be typed directly.

Common uses

- Insert a computed intermediate value into a worked solution line.
- Insert a table or figure generated by helper functions.
- Use a specialized PG helper when PGML does not provide a direct syntax.

Example: computed text inside math mode

```
BEGIN_PGML
To solve [: x + [$a] = [$b] x :], subtract [:x:] from both sides to get
[: [$a] = ([[$b]-1) x = [@ $b - 1 @] x :].
END_PGML
```

HTML in variables is escaped by default

If you store HTML markup in a Perl variable (for example, `$label = "<span style='color:red'>warning</span>"`) and insert it with `[$label]`, PGML will escape the angle brackets and the student will see the raw tags instead of styled text.

To pass HTML through unescaped, append an asterisk:

```
[$label]*
```

This tells PGML to treat the variable's output as pre-formatted HTML, the same trailing `*` used after `[@ ... @]*` eval blocks. Use it whenever a variable contains `<span>`, `<div>`, `<sup>`, `<sub>`, or any other HTML tags that should render as markup.

Avoid MODES() inside eval blocks

A common mistake is to use `MODES(HTML => '...')` inside a `[@ ... @]` eval block, expecting it to emit raw HTML. In practice, `MODES(...)` returns `1` in eval context, so the student sees the number 1 instead of the intended content.

If you need mode-dependent output, build it in the Perl setup section and insert the result with `[$var]*`, or use PGML tag wrappers and raw HTML outside eval blocks.

MathJax color commands

TeX color macros such as `\color{}` and `\textcolor{}` do not reliably render in PGML because MathJax may wrap them in `tex2jax_ignore` spans. If you need colored text in a prompt, use HTML `<span style="color:...">` instead. For a fuller discussion of MathJax rendering constraints, see [Text Coloring and Emphasis](#).

Blocked HTML tags

PGML's HTML whitelist does not allow raw `<table>`, `<tr>`, `<td>`, or `<th>` tags. If you write them in PGML output they will be stripped or trigger warnings. Use `niceTables.pl` for any tabular layout. For more on the tag whitelist and workarounds, see [Making Tables with niceTables](#).

Images and links (LibreTexts constraint)

Images can be inserted via helper functions and command substitution, but the textbook content in this repository avoids HTML links because LibreTexts provides its own table of contents and in-platform navigation. Keep prompts self-contained and do not rely on linking out to other pages.

Treat this as an advanced escape hatch: keep substitutions small, readable, and easy to test.

Next, depending on what you are doing:

- If you only need a data table, go to [Tables and structured data](#).
- If you need a blank that grades correctly, go to [Answer blanks and answers](#).
- If you are writing a workflow prompt, go to [Lists and workflows](#).

This page titled [3.6: Command substitution and inserts](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

4: Breaking Down the Components of a WeBWork Problem

Index: section, what it controls, and what to check when debugging.

Section	What it controls	What to check when debugging
OPL header	Metadata, searchability, and attribution.	Missing TITLE/DESCRIPTION or empty tags.
Preamble	Macros, rendering switches, and global knobs.	Missing macro files, wrong load order, or disabled features.
Setup	Values, constraints, answers, and graders.	Choice list mismatches, wrong correct answer, or grading mistakes.
Statement	PGML prompt text, structure, and blanks.	Blank binding, missing inserts, or formatting breaks.
Solution	Explanations, trust, and grading alignment.	Mismatch between solution conventions and the checker.

Chapter 4 is a single worked problem sliced into OPL Header, Preamble, Setup, Statement, Solution, and a Put it together workflow page. It mirrors the OpenWeBWork PG sample-problem structure while acknowledging the five-section reality so you can scan each layer quickly and edit the right section first.

This chapter is not teaching new syntax. It is teaching how to read a working WeBWork problem so you know where to edit safely.

The five parts of a problem are visually distinguished throughout this chapter so you can spot where an edit belongs at a glance.

Running example: a short multiple-choice prompt with two radio-button blanks, one vertical and one horizontal.

Instead of treating a problem as a block of code, this chapter treats it as five parts that each have a job and each fail in predictable ways. The goal is not to memorize syntax, but to build a reliable editing habit: change one part, test it quickly, and keep the file readable for the next person.

One-screen anatomy (table version)

One-screen anatomy (table version) reference.

Section name	What readers usually change	What usually breaks
OPL header	TITLE, keywords, DBsubject tags.	Missing DESCRIPTION or empty tags.
Preamble	Macro list, display options, parser features.	Missing macro file, wrong load order, or missing feature.
Setup	Answer objects, choice lists, and correctness rules.	Choice list mismatches or a wrong correct answer.
Statement	Story wording, tables, blanks, formatting.	Unbound blanks, missing variables, malformed PGML.
Solution	Explanation steps, rounding notes, units.	Solution shows a different convention than the checker.

If ADAPT reports only a generic error, do a fast local render from [Testing and Debugging](#) to get a line number before you troubleshoot the section tables below.

Section pages

- [Full file](#) – Full file (reference copy)
- [OPL Header](#) – OPL Header
- [Preamble](#) – Preamble

- [Setup](#) – Setup
- [Statement](#) – Statement
- [Solutions](#) – Solution
- [Putting It All Together](#) – Putting it together

Practical edit map

Practical edit map: task, where to edit, and fast check.

Task	Where to edit	Fast check
Change options	Setup (choice lists)	Render and confirm the correct label still matches.
Change prompt wording	Statement (PGML)	Confirm each blank still has an answer attached.
Add another blank	Setup then Statement	Verify the new blank label maps to the right answer.
Change the correct answer	Setup (answer object)	Confirm the Solution shows the same label.

Section snapshots

OPL header

Why this exists: it makes the problem searchable and credits the author.

OPL header: common edits and common failures.

Common edits	Common failures
• Update TITLE or DESCRIPTION. • Adjust DBsubject tags.	• Missing DESCRIPTION block. • Empty or mismatched tags.

Preamble

Why this exists: it declares the macros and global options that make the rest of the file possible.

Preamble: common edits and common failures.

Common edits	Common failures
• Add or remove macro files. • Enable a parser or display option.	• Macro file not loaded. • Macro load order hides a feature.

Setup

Why this exists: it defines the values, constraints, and answer objects that the PGML uses.

Setup: common edits and common failures.

Common edits	Common failures
• Edit choice lists or correct answers. • Adjust layout or ordering options.	• Correct answer no longer matches the list. • Statement points to the wrong answer object.

Statement

Why this exists: it is the PGML prompt students read, including data, structure, and blanks.

Statement: common edits and common failures.

Common edits	Common failures
• Rewrite the story or add a data table. • Insert or remove answer blanks.	• Blank missing its attached answer. • Variable used before it is defined.

Solution

Why this exists: it shows the grading target and builds trust without becoming a second lecture.

Solution: common edits and common failures.

Common edits	Common failures
• Clarify the correct option or rationale. • Explain how the choice is determined.	• Solution names a different option. • Solution label does not match grading.

[4.1: Full file](#)

[4.2: OPL Header](#)

[4.3: Preamble](#)

[4.4: Setup](#)

[4.5: Statement](#)

[4.6: Solution](#)

[4.7: Putting It All Together](#)

This page titled [4: Breaking Down the Components of a WeBWorK Problem](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.1: Full file

Full file: section, what it controls, and what to check when debugging.

Section	What it controls	What to check when debugging
OPL header	Metadata, searchability, and attribution.	Missing TITLE/DESCRIPTION or wrong DBsubject tags.
Preamble	Macros, rendering switches, and global knobs.	Missing macro files, wrong load order, or disabled features.
Setup	Values, constraints, answers, and graders.	Bad ranges, unit mismatches, or tolerance mistakes.
Statement	PGML prompt text, structure, and blanks.	Blank binding, missing inserts, or formatting breaks.
Solution	Explanations, trust, and grading alignment.	Mismatch between solution conventions and the checker.

This page shows the full problem file as a reference copy so you can see the entire flow from top to bottom. For an overview of the five sections, see [Sections within a PG question](#). The prompt text uses [PGML](#). Read the file once as a story: macros are loaded, values are chosen, the prompt is written, answers are graded, and a solution explains the work.

When you start editing, avoid making changes on this page directly. Use the section pages to make one targeted change at a time, then come back here to confirm that the full file still reads cleanly and that values and answers are consistent throughout.

As you read, notice which lines are structural and which lines are content. Structural lines are macro loading and section delimiters; content lines are the prompt text, choice lists, and the explanation in the solution.

Section anchors (reference map)

Use this table to find the line where each section starts and ends. When debugging, start by identifying which section contains the error, then jump to the section-specific page for detailed guidance.

Section anchors (reference map): section, starts at, ends at, and more.

Section	Starts at	Ends at	One-line description
OPL header	## TITLE(...)	## Institution(...)	Metadata that makes the problem searchable.
Preamble	# ===== PREAMBLE =====	# ===== SETUP =====	Loads macros and switches that unlock features.
Setup	# ===== SETUP =====	# ===== STATEMENT =====	Defines values, constraints, and answer objects.
Statement	# ===== STATEMENT =====	# ===== SOLUTION =====	Contains the PGML prompt students read.
Solution	# ===== SOLUTION =====	ENDDOCUMENT();	Explains the target and matches the grading rules.

Each section page (4.2 through 4.6) walks through the purpose, common edits, and failure patterns for that part of the file. Start with the section closest to your error. This page is a reference copy so you can see the whole file in one place after you make changes in the section-specific pages.

The tables below give you a one-line summary of what each section does, what you typically edit there, and what fails most often.

Section quick reference

OPL header

Why this exists: it documents metadata for search and attribution before any code runs.

OPL header: common edits and common failures.

Common edits	Common failures
Update TITLE, KEYWORDS, and DBsubject tags.	Missing DESCRIPTION or empty tags.

Preamble

Why this exists: it declares the macros and global options that make the file render.

Preamble: common edits and common failures.

Common edits	Common failures
Add or remove a macro file.	Missing macro that defines a feature.

Setup

Why this exists: it defines the answer objects and correct choices.

Setup: common edits and common failures.

Common edits	Common failures
Edit choice lists or correct answers.	Correct answer no longer matches the list.

Statement

Why this exists: it is the PGML prompt with readable structure and blanks.

Statement: common edits and common failures.

Common edits	Common failures
Rewrite the story or add a table.	Blank not bound to an answer.

Solution

Why this exists: it explains the grading target and the method.

Solution: common edits and common failures.

Common edits	Common failures
Clarify the correct option or rationale.	Solution label does not match grading.

Complete file (reference copy)

This is the complete worked example assembled from sections 4.2 through 4.6. Read it once as a story -- macros load, values are chosen, the prompt appears, answers are graded, and the solution explains the work.

```
# ===== OPL HEADER =====  
## TITLE('Color choice using radio buttons')  
## DESCRIPTION  
## Select the correct color label from two radio button formats and confirm the choice  
## ENDDescription  
## KEYWORDS('radio buttons','choice list','PGML prompt')
```

```
## DBsubject('Biochemistry')
## DBchapter('Lab techniques')
## DBsection('Reporting conventions')
## Date('2026-01-28')
## Author('Example author')
## Institution('Example institution')

# ===== PREAMBLE =====
DOCUMENT();

loadMacros(
  "PGstandard.pl",
  "PGML.pl",
  "parserRadioButtons.pl",
  "PGcourse.pl",
);

# ===== SETUP =====
$radio1 = RadioButtons(
  [ 'Red', 'Blue', 'Green', 'None of these' ],
  'Blue',
);

$radio2 = RadioButtons(
  [ 'Red', 'Blue', 'Green', 'None of these' ],
  2,
  separator => $SPACE x 5
);

# ===== STATEMENT =====
BEGIN_PGML
My favorite color is

[_]{$radio1}

The same question with horizontal options:

My favorite color is [_]{$radio2}
END_PGML

# ===== SOLUTION =====
BEGIN_PGML_SOLUTION
The correct answer to the first is [$radio1->correct_ans].

The correct answer to the second is [$radio2->correct_ans].
END_PGML_SOLUTION
```

```
ENDDOCUMENT ( ) ;
```

This page titled [4.1: Full file](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.2: OPL Header

This section explains the OPL header for the worked example and shows how to pick biology-appropriate classification tags. The header comes before `DOCUMENT()` and acts as the metadata "section zero" for the problem.

Use this page when: you are writing metadata for a new problem or reviewing a problem before publishing.

Worked example header

This is the complete OPL header for the worked example in Chapter 4.

```
## TITLE('Color choice using radio buttons')
## DESCRIPTION
## Select the correct color label from two radio button formats and confirm the choice
## ENDDescription
## KEYWORDS('radio buttons','choice list','PGML prompt')
## DBsubject('Biochemistry')
## DBchapter('Lab techniques')
## DBsection('Reporting conventions')
## Level(2)
## Date('2026-01-28')
## Author('Example author')
## Institution('Example institution')
## Language(en)
```

Customize the TITLE, DESCRIPTION, KEYWORDS, and classification fields to match your problem's content and subject area.

Field checklist

Field checklist: field, what it should contain, and common mistakes.

Field	What it should contain	Common mistakes
TITLE	Short, student-facing task title.	Too long or story-like titles.
DESCRIPTION	2 to 3 sentences describing the task and constraints.	No constraints or overly long explanations.
KEYWORDS	3 to 5 concept or technique keywords.	Story nouns instead of concepts.
DBsubject/DBchapter/DBsection	Life science classification strings.	Leaving these blank or too generic.
Level	Bloom's taxonomy level (1-6).	Omitted or set to wrong cognitive level.
Date/Author/Institution	Attribution fields for reuse.	Omitted attribution.
Language	ISO 639-1 code (e.g., <code>en</code>).	Omitted entirely.

How this header fits into the problem structure

The OPL header is "section zero" - it comes before `DOCUMENT()` and exists outside the traditional four-section structure (Preamble, Setup, Statement, Solution). Think of it as the problem's passport: it makes the problem searchable, classifiable, and reusable in problem banks.

Five-section reality

1. **Section 0: OPL Header** (metadata comments, before `DOCUMENT`)
2. **Section 1: Preamble** (`DOCUMENT` through `loadMacros`)

3. **Section 2: Setup** (Perl code before PGML)
4. **Section 3: Statement** (BEGIN_PGML / END_PGML)
5. **Section 4: Solution** (optional BEGIN_PGML_SOLUTION / END_PGML_SOLUTION)

Traditional PG documentation only lists four sections because it omits the OPL header. However, for long-term reuse and problem bank classification, the header is essential.

Choosing biology classifications

The worked example uses `DBsubject('Biochemistry')` because the problem context is lab-focused. Here's how to choose classifications for other biology problems.

Classification decision guide

Classification decision guide reference.

Problem type	DBsubject	DBchapter suggestion
Dilution calculations, spectrophotometry, gel analysis	Laboratory Techniques	Solution preparation, Protein analysis, DNA analysis
Enzyme kinetics, buffer pH, protein structure	Biochemistry	Enzymes, Acids and bases, Proteins
Punnett squares, pedigrees, linkage	Genetics	Mendelian inheritance, Pedigree analysis, Linkage
Cell signaling, membrane transport, organelles	Cell Biology	Signal transduction, Membranes, Organelles
DNA replication, transcription, translation	Biology - molecular	DNA replication, Gene expression, Translation

Classification guidelines

- **Start with course context:** If the problem is for a Biochemistry course, use `DBsubject('Biochemistry')` even if it involves lab techniques.
- **Use existing tags:** Search your repository for similar problems and copy the exact classification strings to maintain consistency.
- **Be specific in DBsection:** Instead of "General concepts", use "Michaelis-Menten kinetics" or "Serial dilutions".
- **Avoid empty strings:** Never leave DBsubject, DBchapter, or DBsection blank. Choose the closest existing label and revise later if needed.

Quick reference: Header fields explained

TITLE field

Student-facing title that appears in problem listings. Keep it under 80 characters and focus on the task, not the story wrapper. For detailed guidance and good/bad examples, see [OPL Header and Metadata](#).

Example for worked problem: `TITLE('Color choice using radio buttons')`

DESCRIPTION block

Instructor-facing summary (2-3 sentences) that describes the task, constraints (units, rounding), and provenance if adapted from another source. For templates and biology examples, see [OPL Header and Metadata](#).

Example for worked problem: The description mentions "radio button formats" and "choice labels" to clarify what students will interact with.

KEYWORDS field

3-5 technique or concept keywords that make the problem searchable. Avoid story nouns and generic terms. For keyword selection process and biology examples, see [OPL Header and Metadata](#).

Example for worked problem: `KEYWORDS('radio buttons','choice list','PGML prompt')` focuses on interaction type and PGML features, not biology content (since this is a demonstration problem).

Attribution fields

Date, Author, and Institution provide credit and help track provenance for long-term reuse. Always include these fields, even for internal problems.

Textbook provenance fields

When a problem is adapted from a textbook or other source, use these fields to track the original. Number the tags (TitleText1, TitleText2, etc.) to map to multiple texts when appropriate.

```
## TitleText1('Title of source')
## EditionText1('Edition')
## AuthorText1('Author(s)')
## Section1('Section identifier')
## Problem1('Problem identifier')
```

If adapted, also state that in the DESCRIPTION block (e.g., "Adapted from ...").

Level field: Bloom's taxonomy

The `Level(...)` field uses a 1-6 scale aligned to Bloom's revised taxonomy. This field classifies the cognitive demand of the problem and is required.

Level field: Bloom's taxonomy: level, category, and description.

Level	Category	Description
1	Remember	Recall a term, definition, or basic fact. Verbs: recall, define, label, list, identify.
2	Understand	Describe, explain, or classify a concept. Verbs: explain, summarize, interpret, classify.
3	Apply	Use rules or methods in a direct way. Verbs: apply, use, determine, solve.
4	Analyze	Interpret data and connect to conclusions. Verbs: analyze, compare, contrast, infer.
5	Evaluate	Justify choices or critique evidence. Verbs: evaluate, justify, critique, defend.
6	Create	Design or propose something new. Verbs: design, propose, formulate, develop.

The worked example is Level 2 (Understand) because students classify color labels from given options. For the full Bloom's taxonomy scale with biology-specific examples, see [OPL Header and Metadata](#).

Language field

Use ISO 639-1 language codes. This field is required. For English:

```
## Language(en)
```

License comments

Include a license comment block at the top of every problem file for proper attribution and reuse. OPL does not define a standard license tag. Use plain comment lines (single `#`) to state licensing. This is not an OPL tag but is required for the repository.

```
# This work is licensed under CC BY 4.0 (Creative Commons Attribution 4.0
# International License).
```

```
# https://creativecommons.org/licenses/by/4.0/  
# Source code portions are licensed under LGPLv3.
```

Cross-references for detailed guidance

- **OPL Header and Metadata:** Comprehensive OPL header guidance with biology examples, decision trees, and good/bad comparisons.
- **Minimal Templates:** Complete problem templates with pre-filled OPL headers you can copy and adapt.
- **Next sections in Chapter 4:** After the header, see [Preamble](#), [Setup](#), [Statement](#) for how each section builds on the metadata.

Common mistakes in problem context

- **Forgetting to update the header after modifying the problem:** If you change a dilution problem to a genetics problem, update DBsubject, DBchapter, KEYWORDS, and DESCRIPTION.
- **Copying header from unrelated problem:** Always customize TITLE, DESCRIPTION, and classification tags to match the new problem's actual content.
- **Smart quotes in header:** Use plain ASCII quotes ('), not curly quotes (' or '). Smart quotes break OPL parsing tools.
- **Empty classification fields:** Never leave DBsubject, DBchapter, or DBsection empty. Pick the closest existing tag and revise later if needed.

Apply it today

- Check that every problem has TITLE and DESCRIPTION before you edit the prompt.
- Update the DBsubject tags whenever the scientific context changes.
- Use existing classification strings from your repository to maintain consistency.
- Cross-reference [OPL Header and Metadata](#) for detailed field guidance and biology examples.
- Use templates from [Minimal Templates](#) for pre-filled OPL headers.

This page titled [4.2: OPL Header](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.3: Preamble

Preamble

The Preamble is the environment setup for the problem. It loads the macro files that make PGML work and unlocks specific interaction patterns the rest of the file assumes exist. For a reference list of commonly used macros, see [Common PG Macros](#). For background on PG files, see [PG Problem Files](#).

When a problem fails to load with a generic error, the cause is almost always a missing or mismatched macro in the preamble. This section is boring when it works and unforgiving when it does not, so most authors touch it last, not first.

How to read this code

When reading the preamble, ignore the story of the problem entirely. This section exists to answer one question only: what tools does this problem need in order to render and grade correctly. If something fails to load, displays as plain text, or shows missing widgets, the cause is almost always here.

The excerpt below is the exact preamble used in the worked example. It shows the macro files that are loaded before any values or text are defined.

```
DOCUMENT();

loadMacros(
  "PGstandard.pl",
  "PGML.pl",
  "parserRadioButtons.pl",
  "PGcourse.pl"
);
```

Line notes

How to read this code reference.

Line or snippet	What it does	What you would change first
DOCUMENT();	Initializes the problem file so WeBWorK knows where it begins.	Almost never changed.
PGstandard.pl	Provides core PG functionality used by nearly all problems.	Always required.
PGML.pl	Enables PGML formatting and answer blanks.	Remove only if converting away from PGML.
parserRadioButtons.pl	Enables radio button answer widgets.	Add or remove when changing interaction type.
PGcourse.pl	Loads course-level helpers and defaults.	Usually left alone.

A common mistake is copying an example from elsewhere and missing a macro that enables a widget. When that happens, the problem often renders with no useful error message unless you test locally.

Why this exists: it declares which macro files and features the problem can use.

How to read this code: common edits and common failures.

Common edits	Common failures
- Add a macro to enable a new interaction. - Remove a macro that is not used.	- A feature is missing because the macro is not loaded. - Load order masks a macro or parser option.

Macro selection (what to load)

Macro selection (what to load) reference.

Macro	What it enables	When you need it	Common failure	Canonical example
PGstandard.pl	Core PG functions and helpers.	Always (baseline PG utilities).	Undefined basic functions.	OpenWeBWorK PG samples: most core examples.
PGML.pl	PGML prompt blocks and inserts.	Any PGML-first prompt.	BEGIN_PGML fails or renders raw text.	OpenWeBWorK PG samples: PGML intro pages.
parserRadioButtons.pl	Radio-button answer widgets.	Single-correct multiple choice prompts.	Radio buttons do not render.	OpenWeBWorK PG samples: radio button techniques.
PGcourse.pl	Course macros and shared helpers.	You rely on course-wide helpers.	Custom helper not found.	OpenWeBWorK PG samples: course macro usage.

For radio-button interactions, load `parserRadioButtons.pl` and mirror the official radio button technique pages in the OpenWeBWorK PG samples so the UI matches the pattern you want.

Useful reference samples include `parserMultiAnswer` and `NumericalTolerance` in the OpenWeBWorK PG samples, which show canonical patterns for coordinated blanks and tolerance handling when you move beyond single-choice prompts.

Some older examples use legacy macros or interaction widgets that still appear online; prefer the patterns shown in the official PG sample problems when choosing an interaction style.

When it breaks

When it breaks: symptom, likely cause, and fix.

Symptom	Likely cause	Fix
Macro not found error	Missing macro file in <code>loadMacros()</code> .	Add the macro and keep load order consistent.
Choice widget renders as plain text	Radio button macro file not loaded.	Load <code>parserRadioButtons.pl</code> in the Preamble.
PGML block not recognized	Missing <code>PGML.pl</code> .	Load <code>PGML.pl</code> before the PGML block.

Broken example: missing macro for the interaction

This problem fails before the statement even renders. The author copied a radio button example but did not load the macro that provides the widget. In ADAPT, this often appears as a generic error with no line number.

```
DOCUMENT ( );

loadMacros(
    "PGstandard.pl",
    "PGML.pl",
```

```
"PGcourse.pl"  
);
```

Later in the file, the setup uses `RadioButtons` .

What went wrong

- The interaction widget exists in setup, but the macro that defines it was never loaded.
- The PG file parses, but rendering fails when the widget is encountered.
- ADAPT typically reports a generic error loading the problem.

How to recognize it

- The file worked in another problem you copied from.
- The error appears before any text renders.
- Local rendering reports an undefined subroutine or missing macro.

How to fix it

Add the missing macro to `loadMacros()` and re-render immediately. This is why preamble problems should always be tested locally before import.

This page titled [4.3: Preamble](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.4: Setup

Setup

Setup is where a general prompt becomes a specific, randomized instance. This is where you define values, choose realistic options, and create answer objects for the blanks in the Statement. For randomization patterns and best practices, see [Advanced Randomization Techniques](#). Variables defined here are inserted into the prompt using [command substitution](#). For a catalog of question types and the widgets they use, see [Different Question Types](#).

Setup is the section most life science instructors edit. Think of it as the lab notebook for the problem: if the values, units, or tolerances are wrong here, no amount of prompt editing will fix grading.

How to read this code

The setup section is where the abstract idea of the problem becomes concrete. This is where values are chosen, constrained, and combined into the exact targets the grader will check. If a problem behaves differently across seeds or produces surprising answers, this is the first place to look.

The excerpt below defines the answer objects used later in the statement. Nothing here is visible to students, but everything here controls grading behavior.

```
$radio1 = RadioButtons(
    [ 'Red', 'Blue', 'Green', 'None of these' ],
    'Blue',
);

$radio2 = RadioButtons(
    [ 'Red', 'Blue', 'Green', 'None of these' ],
    2,
    separator => $SPACE x 5
);
```

Line notes

How to read this code reference.

Line or snippet	What it does	What you would change first
<code>RadioButtons(...)</code>	Creates a radio-button answer object.	Change this if you switch interaction types.
<code>['Red', 'Blue', 'Green', 'None of these']</code>	Defines what students can select.	Edit when changing the story or options.
<code>'Blue'</code>	Sets the correct answer by value.	Use this when clarity matters more than indexing.
<code>2</code>	Sets the correct answer by index (starting at 0).	Use when randomizing or reordering choices.
<code>separator => \$SPACE x 5</code>	Controls layout spacing.	Adjust for readability only.

A frequent source of confusion is changing the prompt text without updating the corresponding answer object. Keeping all answer definitions together in setup makes these mismatches easier to spot.

Why this exists: it turns a story into answer objects and grading rules that the PGML can use.

How to read this code: common edits and common failures.

Common edits	Common failures
- Edit choice lists or correct answers. - Adjust layout spacing or ordering.	- Correct answer no longer matches the choice list. - Statement references the wrong answer object.

Try it now: change one choice, render a variant, and confirm the correct answer still matches the intended option and the layout is readable.

Answer objects and choices

Answer objects and choices reference.

Answer object	Choice list	Correct answer	Why it exists
radio1	Red, Blue, Green, None of these	Blue (by value)	Single-choice prompt with a clear correct label.
radio2	Red, Blue, Green, None of these	Index 2	Same options with a horizontal layout.

Answer targets are the glue between Setup and Statement: these are the answer objects attached to blanks in the prompt.

Answer targets

Answer targets: answer object, correct answer, how it is set, and more.

Answer object	Correct answer	How it is set	Used by which blank
radio1	Blue	Value string	radio1 blank
radio2	Index 2	Index in list (0-based)	radio2 blank

Answer object options

Answer object options: option, what it controls, and when to use it.

Option	What it controls	When to use it
Correct answer by value	Chooses the correct option by its label.	When clarity matters more than index tracking.
Correct answer by index	Chooses the correct option by position.	When you randomize or reorder choices.
separator => \$SPACE x 5	Spacing between choices.	When horizontal layout needs breathing room.

Broken example: correct answer does not match the story

This problem renders and accepts input, but grading behaves unexpectedly. Students select the correct option according to the prompt, but the system marks it wrong.

```
$radio = RadioButtons(
    [ 'ATP', 'ADP', 'AMP' ],
    'AMP',
);
```

The statement later asks which molecule has the highest energy.

What went wrong

- The story implies ATP is correct.
- The setup defines AMP as the correct answer.
- The grader is doing exactly what it was told.

How to recognize it

- Multiple students report being marked wrong for the same choice.
- Previewing the problem shows consistent wrong grading.
- The solution displays a different answer than the prompt implies.

How to fix it

Always read the setup and the statement back to back. When changing the story, update the setup first, then confirm the statement still matches.

Keep the choice lists and correct answers aligned with the statement text so the grading intent stays obvious.

This page titled [4.4: Setup](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.5: Statement

Statement

The Statement is the student-facing prompt written in [PGML](#). It is where the story, data, and tasks become visible, and where each blank must be tied to one answer object from Setup.

The Statement never grades anything. Blanks are just hooks into Setup, and PGML reads top to bottom like a worksheet, not like code. If something grades incorrectly, the cause is almost never here, even if the formatting looks scary.

How to read this code

The statement is the only part of the file that students actually read. Your goal here is to keep the text readable while making sure every blank is unambiguous and attached to the correct answer checker. PGML allows you to do this without mixing formatting logic into setup code.

The excerpt below shows the full statement used in the example, including how answer objects are inserted into the text.

```
BEGIN_PGML
My favorite color is

[_]{$radio1}

The same question with horizontal options:

My favorite color is [_]{$radio2}
END_PGML
```

Line notes

How to read this code reference.

Line or snippet	What it does	What you would change first
BEGIN_PGML	Starts PGML rendering.	Required for PGML formatting.
Plain text	Displays directly to students.	Edit freely for clarity.
[_]{\$radio1}	Inserts a blank bound to an answer checker.	Reorder or duplicate blanks here.
{\$radio2}	References a second answer object.	Update if setup changes.
END_PGML	Ends the PGML block.	Always required.

If grading works but students are confused, the problem is usually here rather than in setup. Read the statement out loud and check whether each blank clearly signals what kind of answer is expected.

Think of the Statement as a contract. The text you show here must match what the checker accepts, or students will feel the grading is arbitrary even when their reasoning is right. Example: the prompt asks for one choice, but the blank is tied to a different answer object.

Choose the structure that reduces reading load: use short paragraphs for a single task, lists for steps, and tables for multi-value data. For emphasis and color options, see [Text Coloring and Emphasis](#). If a prompt feels hard to scan, structure is usually the fix, not more words.

Why this exists: it is the student-facing prompt that ties the story to the answer blanks.

How to read this code: common edits and common failures.

Common edits	Common failures
- Rewrite the story in plain language. - Reorder blanks or add a second prompt variant.	- Blank not attached to the intended answer object. - Answer object name does not match setup.

Fast test: render one seed, enter the expected answer, confirm grading, then change the seed and repeat.

Blanks map

Blanks map: blank label, what it asks for, answer object, and more.

Blank label	What it asks for	Answer object	Student format expectations
radio1	Single-choice selection	\$radio1	Select one option.
radio2	Single-choice selection (horizontal)	\$radio2	Select one option.

Blank binding smell test: if a blank is visible but always marks wrong, confirm it is attached to an answer object defined in Setup.

Broken example: blank attached to the wrong answer object

This is one of the most common PGML mistakes. The problem renders cleanly, but answers are silently mismatched.

```
BEGIN_PGML
The enzyme with the highest activity is [_]{$radio2}.
END_PGML
```

In setup, `$radio1` and `$radio2` are both defined, but `$radio2` corresponds to a different question.

What went wrong

- The blank is bound to the wrong answer object.
- The prompt text and grading logic no longer refer to the same concept.
- There is no syntax error, so nothing warns the author.

How to recognize it

- The blank accepts input, but grading feels random.
- Swapping answers between blanks changes which one grades correctly.
- Reviewing setup reveals multiple answer objects with similar names.

How to fix it

Use consistent naming and keep the statement immediately adjacent to the setup during editing. When in doubt, temporarily print `->correct_ans` in the solution to confirm alignment.

Use tables only when there is real data to scan or compare, not to simulate layout. Lists and short paragraphs are usually better for narrative prompts.

Structure patterns

Structure patterns: pattern, when to use, and avoid when.

Pattern	When to use	Avoid when
Short paragraphs	One clear story with 1 to 2 tasks.	Steps are procedural or multi-part.
Lists	Protocols or multi-step directions.	The prompt is a single sentence.
Tables	Real data or comparisons need scanning.	Two values can be in plain text.
Inline math	Small substitutions and unit checks.	Equations are long or multi-line.

For radio buttons and other choice-based inserts, load `parserRadioButtons.pl` and follow the OpenWeBWorK technique samples for multiple choice patterns, using those pages as the pattern directory.

MultiAnswer is an optional advanced variant for interdependent blanks; use the `parserMultiAnswer.pl` documentation and its sample problem when you need shared logic.

This page titled [4.5: Statement](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.6: Solution

Solution

The Solution is for trust and alignment, not grading. It shows students what the system expected and helps instructors confirm that the checker matches the stated method.

The solution does not affect grading, but students trust it more than the grader. A correct grader with a confusing solution still feels broken, and any mismatch creates disputes.

How to read this code

The solution is not a second version of the prompt. It is a short explanation that mirrors the grading logic so students can reconcile their work with what the system checked. Solutions use the same [PGML syntax](#) as the statement block. A good solution is compact, accurate, and consistent with the setup section.

The excerpt below shows the solution block used in the example.

```
BEGIN_PGML_SOLUTION
The correct answer to the first is [$radio1->correct_ans].

The correct answer to the second is [$radio2->correct_ans].
END_PGML_SOLUTION
```

Line notes

How to read this code reference.

Line or snippet	What it does	What you would change first
BEGIN_PGML_SOLUTION	Starts the solution block.	Required for PGML solutions.
Explanatory text	Describes the correct reasoning.	Keep aligned with grading.
->correct_ans	Retrieves the expected answer.	Useful for debugging.
Inline PGML	Formats output cleanly.	Match statement style.
END_PGML_SOLUTION	Ends the solution block.	Always required.

If the solution disagrees with grading, students will trust the grader and distrust the explanation. Keep the solution minimal and mechanically consistent with the setup logic.

The solution must use the same conventions the checker uses. If the checker expects the label "Blue," the solution should show "Blue," not a shorthand label or a different ordering.

Why this exists: it clarifies the target, the method, and the reporting format that the grader expects.

How to read this code: common edits and common failures.

Common edits	Common failures
- State the correct option in plain language. - Add a short rationale for the choice.	- Solution shows a different label than the checker. - Solution implies a different set of choices.

Solution checklist

Solution checklist: include, why, and one-sentence example.

Include	Why	One-sentence example
Restate the prompt	Confirms the student is using the same choices.	"The choices are Red, Blue, Green, or None of these."
State the correct choice	Makes the target explicit.	"The correct answer is Blue."
Short rationale	Connects the answer to the intent.	"Blue matches the prompt's condition."
Exact label	Prevents grading mismatches.	"Use the same label shown in the options list."
Format reminder	Keeps expectations clear.	"Select a single option."

Debugging alignment

Use this table as a quick alignment check: the solution should echo the same conventions the checker uses so students see the expected format.

Debugging alignment: grading expectation and solution alignment.

Grading expectation	Solution alignment
Correct answer by value	Use the exact label from the choice list.
Correct answer by index	Show the label, not the index.
Reordered choice list	Mirror the displayed order or name the label.

Helpful feedback hooks (optional)

Helpful feedback hooks (optional): hook and why it helps.

Hook	Why it helps
Short rationale line	Reinforces why the correct option is correct.
Distractor reminder	Helps students distinguish near-miss options.
Format reminder	Signals that exactly one choice is expected.

Broken example: solution contradicts grading logic

This problem grades correctly, but the solution confuses students. The explanation no longer matches what the system actually checked.

```
BEGIN_PGML_SOLUTION
The correct answer is ATP because it has the highest energy.
END_PGML_SOLUTION
```

In setup, the answer checker expects ADP.

What went wrong

- The solution was written before the setup was finalized.
- The grading logic changed, but the explanation did not.
- Students see a contradiction between feedback and results.

How to recognize it

- Students say, "The solution says X but I was marked wrong for X."
- Preview mode shows correct grading with an incorrect explanation.
- Instructors trust the solution text instead of the grader.

How to fix it

Treat the solution as a mirror of the grading logic, not an independent explanation. When debugging, temporarily echo the computed answer in the solution to verify consistency.

This page titled [4.6: Solution](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

4.7: Putting It All Together

Pick the change, edit the right section, render one seed to confirm grading, then sweep a few seeds to catch edge cases. For the full catalog of question types, see [Different Question Types](#). For macro reference details, see [Common PG Macros](#). Most failures trace to missing macros, bad ranges, unbound blanks, or a mismatch between prompt and checker.

When something goes wrong, the goal is not to fix the whole problem at once. The goal is to identify which section owns the failure, edit only that section, and render early and often before you return to ADAPT.

Putting it together reference.

Section	What it controls	What to check when debugging
OPL header	Metadata, searchability, and attribution.	Missing TITLE/DESCRIPTION or empty tags.
Preamble	Macros, rendering switches, and global knobs.	Missing macro files, wrong load order, or disabled features.
Setup	Values, constraints, answers, and graders.	Choice list mismatches, wrong correct answer, or grading mistakes.
Statement	PGML prompt text, structure, and blanks.	Blank binding, missing inserts, or formatting breaks.
Solution	Explanations, trust, and grading alignment.	Mismatch between solution conventions and the checker.

Common edits (workflow map)

Common edits (workflow map) reference.

What you want to change	Edit here first	Then verify here	Fast test
Update metadata or subject tags	OPL header	Full file	Confirm tags and description are filled.
Swap interaction type	Preamble	Statement	Check that the widget renders.
Change options or the correct answer	Setup	Solution	Confirm the correct label still matches.
Add a new blank	Setup	Statement	Confirm the blank shows the right answer.
Explain why the choice is correct	Solution	Statement	Check wording against the choice list.

Preflight before ADAPT import

ADAPT errors are often generic, so local rendering is the first diagnostic step when something fails to preview.

Preflight before ADAPT import: check and why it matters.

Check	Why it matters
Render locally (Testing and Debugging) if ADAPT gives only a generic error.	Line numbers save time when a macro or PGML issue appears.
Preview 3 to 5 seeds.	Catches impossible variants early.
Scan option labels in prompt, blanks, and solution.	Prevents silent grading mismatches.

When you want to swap interaction types, use the techniques list in the OpenWeBWorK PG samples as your menu of proven patterns.

Complete example: biology RadioButtons problem

This complete problem demonstrates all five sections (OPL header, preamble, setup, statement, and PGML prompt) working together. Copy this template and modify the content for your own biology questions.

```
## TITLE('Identify an amino acid property')
## DESCRIPTION
## Select the amino acid that is nonpolar from a short list.
## Demonstrates RadioButtons with biology content.
## ENDDescription
## KEYWORDS('amino acids','nonpolar','RadioButtons')
## DBsubject('Biochemistry')
## DBchapter('Amino Acids')
## DBsection('Classification')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$rb = RadioButtons(
    [ 'Leucine', 'Glutamate', 'Lysine', 'Serine' ],
    'Leucine',
    labels      => 'ABC',
    displayLabels => 1,
);

BEGIN_PGML
Which of the following amino acids has a nonpolar
(hydrophobic) side chain?

[_]{$rb}
END_PGML

ENDDOCUMENT();
```

This page titled [4.7: Putting It All Together](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

5: Different Question Types in WeBWorK

This chapter is the bridge between WeBWorK authoring and using WeBWorK inside ADAPT. It collects common question interaction patterns (question types) and pairs them with a practical workflow and a short QA checklist so you can ship reliable assignments without surprises.

Most biology and biochemistry prompts start as a story with a single decision, a short label, or a short workflow. If the grading needs robust math parsing, multiple equivalent forms, or algorithmic variants, use PGML-first WeBWorK. If the task is recognition, labeling, or a short protocol step check, these interaction types are usually faster to build and easier to maintain.

Instructors usually lose time in two places: blank binding mistakes and generic error testing loops. Use the [workflow and QA page](#) as the default finishing step, even when a question looks correct in author view.

WeBWorK problems are authored using PG (a Perl-based setup layer) and PGML (a markup layer for the student-facing prompt). In this guide, PG is treated as minimal scaffolding and PGML is the default authoring style.

Life-science-first emphasis: examples and checklists are written to answer "what breaks when the content is biology?" and to prioritize patterns that grade reliably for biology-style prompts. For advanced techniques such as colored emphasis, table layouts, and matching with two-column displays, see [Advanced PGML Techniques](#).

[5.1: Question Types Overview](#)

[5.2: Multiple Choice](#)

[5.3: Multiple Answer](#)

[5.4: Matching](#)

[5.5: Numerical Entry](#)

[5.6: Fill in the Blank](#)

[5.7: Multipart Questions](#)

[5.8: Ordered List](#)

[5.9: Workflow and QA](#)

This page titled [5: Different Question Types in WeBWorK](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.1: Question Types Overview

This section introduces core question types used across many assessment systems. These are often useful for concept checks, terminology, and simple workflows that do not require WeBWorK-style answer parsing or algorithmic randomization.

In this textbook, question types are treated as a practical authoring skill: you learn the patterns in Chapter 5, and then [Advanced PGML Techniques](#) shows how to apply those patterns in specific scientific subjects.

Use Multiple Choice when there is one best scientific claim and you can write distractors that represent real misconceptions. Use Multiple Answer when the concept is a set and you can define the set boundaries clearly. Use Matching for term-to-meaning mapping, Fill in the Blank for short labels with controlled variants, and Ordered List only when the sequence is truly standard in your context.

Life science questions fail in predictable ways: naming conventions drift, unit prefixes drift, and prompts quietly allow more than one interpretation. When that risk is high, prefer recognition-style items with explicit wording, or switch to WeBWorK answer checking where you can enforce structure.

How this fits a PGML-first course

- Use WeBWorK (PGML-first) for numeric and formula-based practice where robust parsing and answer checking matters.
- Use these question types for portable, simple interaction patterns (multiple choice, matching, ordering, fill-in-the-blank).

Supported question types

- [Multiple Choice \(MC\)](#)
- [Multiple Answer \(MA\)](#)
- [Matching \(MATCH\)](#)
- [Numerical Entry \(NUM\)](#)
- [Fill-in-the-Blank \(FIB\)](#)
- [Multi-Part Fill-in-the-Blank \(MULTI_FIB\)](#)
- [Ordered List \(ORDER\)](#)

Science prompt style (recommended)

- Keep the story stable and short.
- Use explicit numbers and units in the prompt when relevant.
- Prefer tables for data instead of long prose when the problem is data-driven.
- For concept checks, write distractors that reflect common misconceptions.

Input conventions

Each type is defined by a small set of inputs. The next sections show a science example for each type using those inputs.

This page titled [5.1: Question Types Overview](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.2: Multiple Choice

Multiple Choice (MC) is best for a single conceptual decision where one answer is clearly correct and the distractors represent plausible misconceptions.

A good biology multiple choice item is really a single claim test. Write the stem so the correct choice is correct for one reason, and each distractor is wrong for one specific reason you would actually see on an exam. If two choices are defensible under different assumptions, the problem is the stem, not the students.

The most common break is mixed formats inside the choice list, like fractions mixed with decimals or words mixed with symbols. Keep choices parallel and decide your conventions first, then write distractors to match that convention.

Inputs

- `question_text` (str): the question prompt
- `choices_list` (list): answer choices
- `answer_text` (str): the correct answer

Science example (genetics)

Use this type for a single decision, like recognizing a pattern in an inheritance scenario.

```
question_text:
  In a diploid organism, allele A is dominant to allele a.
  Two heterozygous parents (Aa x Aa) produce offspring.
  What fraction of offspring are expected to be homozygous recessive (aa)?

choices_list:
  - 1/4
  - 1/2
  - 3/4
  - 1

answer_text: 1/4
```

Official example (OpenWeBWorK PG samples): [MultipleChoiceRadio](#).

Preferred PGML pattern

Use `RadioButtons` with `parserRadioButtons.pl` (see [Common PG Macros](#) for macro loading) and attach the evaluator inline in the PGML blank. This is the standard pattern for all multiple choice questions. For statement-by-statement evaluation patterns, see [Randomized MC True and False Statements](#).

```
loadMacros(
  "PGstandard.pl",
  "PGML.pl",
  "parserRadioButtons.pl",
  "PGcourse.pl",
);

$rb = RadioButtons(
  [ [ 'Red', 'Blue', 'Green' ], 'None of these' ],
```

```
'Blue',
labels      => 'ABC',
displayLabels => 1,
randomize   => 0,
);

BEGIN_PGML
My favorite color is

[_]{$rb}
END_PGML
```

Key options:

- `labels => 'ABC'` gives each choice a consistent letter label.
- `displayLabels => 1` shows the letter labels to students.
- `randomize => 0` is the default and keeps the order you defined. Only enable randomization if you are safely mapping the correct answer.
- For horizontal layout, use `separator => $SPACE x 5` or an HTML spacing string.

Complete working examples

Genetics multiple choice -- Identify the expected fraction of homozygous recessive offspring from a monohybrid cross.

```
## TITLE('Monohybrid cross outcome')
## DESCRIPTION
## Determine the expected fraction of homozygous recessive
## offspring from a cross between two heterozygous parents.
## ENDDescription
## KEYWORDS('genetics','monohybrid cross','RadioButtons')
## DBsubject('Genetics')
## DBchapter('Mendelian Genetics')
## DBsection('Monohybrid Cross')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$rb = RadioButtons(
    [ '1/4', '1/2', '3/4', '1' ],
    '1/4',
    labels      => 'ABC',
```

```
        displayLabels => 1,
    );

BEGIN_PGML
In a diploid organism, allele A is dominant to allele a.
Two heterozygous parents (Aa x Aa) produce offspring.
What fraction of offspring are expected to be homozygous
recessive (aa)?

[_]{$rb}
END_PGML

ENDDOCUMENT();
```

Organelle identification -- Identify the organelle responsible for protein synthesis.

```
## TITLE('Identify the protein synthesis organelle')
## DESCRIPTION
## Select the organelle that is the primary site of
## protein synthesis in eukaryotic cells.
## ENDDescription
## KEYWORDS('cell biology','organelles','RadioButtons')
## DBsubject('Cell Biology')
## DBchapter('Organelles')
## DBsection('Function')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$rb = RadioButtons(
    [ 'Ribosome', 'Golgi apparatus', 'Lysosome', 'Smooth ER' ],
    'Ribosome',
    labels      => 'ABC',
    displayLabels => 1,
);

BEGIN_PGML
Which organelle is the primary site of protein synthesis
in eukaryotic cells?
```

```
[_]{ $rb}  
END_PGML  
  
ENDDOCUMENT ( );
```

Authoring notes

- Keep choices parallel in format (all fractions, all numbers, etc.).
- Use distractors that match common mistakes (for example confusing genotype vs phenotype).

This page titled [5.2: Multiple Choice](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.3: Multiple Answer

Multiple Answer (MA) is best when more than one choice is correct and the learning objective includes recognizing a set of correct statements. Unlike [Multiple Choice](#), which expects a single best answer, Multiple Answer requires students to identify all correct items.

Multiple answer works best when you can define membership in the set using a rule students can apply. State "Select all that apply" explicitly, keep the set small, and avoid choices that depend on exceptions you did not teach. This type is excellent for wet lab requirements and "which statements are supported" checks, but it punishes ambiguity.

Inputs

- `question_text` (str)
- `choices_list` (list)
- `answers_list` (list): list of correct answers

Science example (molecular biology)

```
question_text:
  Which of the following are typically required for a standard PCR reaction?
  Select all that apply.

choices_list:
  - Template DNA
  - Primers
  - DNA polymerase
  - Restriction enzyme
  - dNTPs

answers_list:
  - Template DNA
  - Primers
  - DNA polymerase
  - dNTPs
```

Official example (OpenWeBWorK PG samples): [MultipleChoiceCheckbox](#).

Checkbox macro availability

Checkbox macros are unreliable in the PG 2.17 renderer snapshot used by this project (see [Common PG Macros](#) for available macros). Do not use checkbox-style macros unless you have verified the macro exists in the renderer and tested that it renders correctly.

Preferred fallback pattern

Instead of checkboxes, use one `RadioButtons` or `PopUp` widget per statement. Each statement becomes its own True/False or Yes/No decision, and students answer all of them.

```
loadMacros(
  "PGstandard.pl",
  "PGML.pl",
  "parserRadioButtons.pl",
```

```
"PGcourse.pl",
);

$rb1 = RadioButtons([ 'Yes', 'No' ], 'Yes',
  labels => 'ABC', displayLabels => 1);
$rb2 = RadioButtons([ 'Yes', 'No' ], 'Yes',
  labels => 'ABC', displayLabels => 1);
$rb3 = RadioButtons([ 'Yes', 'No' ], 'No',
  labels => 'ABC', displayLabels => 1);

BEGIN_PGML
For each statement, select Yes if it is required for PCR.

Template DNA: [_]{$rb1}

Primers: [_]{$rb2}

Restriction enzyme: [_]{$rb3}
END_PGML
```

Complete working examples

PCR requirements -- Evaluate whether each component is required for a standard PCR reaction using per-statement RadioButtons.

```
## TITLE('Identify PCR requirements')
## DESCRIPTION
## For each component, indicate whether it is required for
## a standard PCR reaction.
## ENDDescription
## KEYWORDS('PCR','molecular biology','true false','RadioButtons')
## DBsubject('Molecular Biology')
## DBchapter('Techniques')
## DBsection('PCR')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
  "PGstandard.pl",
  "PGML.pl",
  "parserRadioButtons.pl",
  "PGcourse.pl",
);

$rb1 = RadioButtons([ 'Yes', 'No' ], 'Yes');
```

```
$rb2 = RadioButtons([ 'Yes', 'No' ], 'Yes');
$rb3 = RadioButtons([ 'Yes', 'No' ], 'Yes');
$rb4 = RadioButtons([ 'Yes', 'No' ], 'No');

BEGIN_PGML
For each component, select Yes if it is required for a
standard PCR reaction.

Template DNA: [_]{$rb1}

Primers: [_]{$rb2}

DNA polymerase: [_]{$rb3}

Restriction enzyme: [_]{$rb4}
END_PGML

ENDDOCUMENT();
```

Enzyme properties -- Evaluate true/false statements about general enzyme characteristics.

```
## TITLE('Enzyme characteristics true or false')
## DESCRIPTION
## Evaluate whether each statement about enzyme properties
## is true or false.
## ENDDescription
## KEYWORDS('enzymes','biochemistry','true false','RadioButtons')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Properties')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$tf = [ 'True', 'False' ];
$rb1 = RadioButtons($tf, 'True');
$rb2 = RadioButtons($tf, 'False');
$rb3 = RadioButtons($tf, 'True');
```

BEGIN_PGML

For each statement, select True or False.

1. Enzymes lower the activation energy of a reaction. ☐{\$rb1}
2. Enzymes are consumed during the reaction they catalyze. ☐{\$rb2}
3. Enzyme activity is affected by temperature and pH. ☐{\$rb3}

END_PGML

ENDDOCUMENT();

Authoring notes

- Say "Select all that apply" explicitly.
- Avoid trick items; keep the set small and clearly defined.

This page titled [5.3: Multiple Answer](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.4: Matching

Matching (MATCH) is best for vocabulary, functional associations, and paired relationships. In science courses, it works well for mapping terms to definitions or steps to outcomes.

Matching is strongest when the task is association, not recall of a sentence from a glossary. Prompts should be short and choices should be short, but the pairing should reflect a functional relationship students use in biology. If the correct pairing is obvious from word shape alone, add one distractor choice only when it forces actual thinking.

Inputs

- `question_text` (str)
- `prompts_list` (list): items to be matched
- `choices_list` (list): possible matching answers

Science example (biochemistry)

```
question_text:
    Match each macromolecule class with the most typical monomer unit.

prompts_list:
    - Proteins
    - Nucleic acids
    - Polysaccharides

choices_list:
    - Amino acids
    - Nucleotides
    - Monosaccharides
```

Official example (OpenWeBWorK PG samples): [Matching](#). Macro reference: [PGchoicemacros](#).

Preferred PGML pattern

Use `PopUp` from `parserPopUp.pl` and place each dropdown inline in PGML next to its prompt. This gives a clean matching layout without legacy PG macros.

```
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserPopUp.pl",
    "PGcourse.pl",
);

$popup1 = PopUp([ '?', 'Amino acids', 'Nucleotides', 'Monosaccharides' ],
    'Amino acids');
$popup2 = PopUp([ '?', 'Amino acids', 'Nucleotides', 'Monosaccharides' ],
    'Nucleotides');
$popup3 = PopUp([ '?', 'Amino acids', 'Nucleotides', 'Monosaccharides' ],
    'Monosaccharides');
```

```
BEGIN_PGML
Match each macromolecule class with its monomer.

1. Proteins [_]{$popup1}
2. Nucleic acids [_]{$popup2}
3. Polysaccharides [_]{$popup3}
END_PGML
```

For advanced matching layouts with tables or formatted columns, see [Randomized Matching Problems](#) (Matching Problems).

Complete working examples

Macromolecule matching -- Match each macromolecule class with its monomer unit using PopUp dropdowns.

```
## TITLE('Match macromolecule classes to monomers')
## DESCRIPTION
## Match each macromolecule class with its monomer unit
## using PopUp dropdown widgets.
## ENDDescription
## KEYWORDS('macromolecules', 'monomers', 'PopUp', 'matching')
## DBsubject('Biochemistry')
## DBchapter('Macromolecules')
## DBsection('Structure')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserPopUp.pl",
    "PGcourse.pl",
);

$popup1 = PopUp(
    ['?', 'Amino acids', 'Nucleotides', 'Monosaccharides'],
    'Amino acids');
$popup2 = PopUp(
    ['?', 'Amino acids', 'Nucleotides', 'Monosaccharides'],
    'Nucleotides');
$popup3 = PopUp(
    ['?', 'Amino acids', 'Nucleotides', 'Monosaccharides'],
    'Monosaccharides');

BEGIN_PGML
Match each macromolecule class with its monomer.
```

```
1. Proteins [_]{$popup1}

2. Nucleic acids [_]{$popup2}

3. Polysaccharides [_]{$popup3}
END_PGML

ENDDOCUMENT();
```

Bond type matching -- Match each molecular scenario with the type of bond or interaction involved.

```
## TITLE('Match molecular interactions to bond types')
## DESCRIPTION
## Match each molecular scenario to the correct bond type
## using PopUp dropdown widgets.
## ENDDescription
## KEYWORDS('chemical bonds','matching','PopUp')
## DBsubject('Chemistry')
## DBchapter('Chemical Bonding')
## DBsection('Bond Types')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserPopUp.pl",
    "PGcourse.pl",
);

$popup1 = PopUp(
    ['?', 'Ionic', 'Covalent', 'Hydrogen'],
    'Covalent');
$popup2 = PopUp(
    ['?', 'Ionic', 'Covalent', 'Hydrogen'],
    'Ionic');
$popup3 = PopUp(
    ['?', 'Ionic', 'Covalent', 'Hydrogen'],
    'Hydrogen');

BEGIN_PGML
Match each molecular scenario to the type of bond
or interaction involved.

1. Two atoms share a pair of electrons [_]{$popup1}
```

2. An electron is transferred from one atom to another [_] { \$popup2 }

3. A partially positive H is attracted to a nearby electronegative atom [_] { \$popup3 }

END_PGML

ENDDOCUMENT () ;

Authoring notes

- Keep the prompts and choices short and parallel.
- If there are obvious one-to-one pairs, add 1-2 extra distractor choices only if it helps learning.

This page titled [5.4: Matching](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.5: Numerical Entry

Numerical Entry (NUM) is best for a single computed value with a defined tolerance. Use this when you want a straightforward numeric result without the full expression parsing features of WeBWorK.

In biology, tolerance is usually about reporting, not instrument precision. Decide what you are grading: the computation, the unit conversion, or the reported number format. Put the unit next to the blank, and pick a tolerance that matches the rounding rule you stated, so the grader is enforcing your instruction, not guessing your intent.

Inputs

- `question_text` (str)
- `answer_float` (float)
- `tolerance_float` (float)
- `tolerance_message` (bool, default=True)

Science example (dilution)

This example uses a simple dilution where students compute a final concentration.

```
question_text:
    You dilute 2.0 mL of a 50.0 mg/mL stock solution to a final volume of 10.0 mL.
    What is the final concentration (mg/mL)?

answer_float: 10.0
tolerance_float: 0.1
tolerance_message: true
```

Official example (OpenWeBWorK PG samples): [NumericalTolerance](#).

Preferred PGML pattern

Use `Compute` with explicit tolerance settings and attach the evaluator inline in the PGML blank.

```
Context("Numeric");
$ans = Compute("10.0")->with(tolType => "absolute", tolerance => 0.1);

BEGIN_PGML
You dilute 2.0 mL of a 50.0 mg/mL stock to 10.0 mL.
What is the final concentration (mg/mL)?

[_____]{$ans}
END_PGML
```

Set up `Context("Numeric")` before defining numerical answers. State the tolerance policy in the question prompt when it matters for grading. For more on answer blanks in PGML, see [Answer blanks and answers](#). For common tolerance mistakes, see [Common Mistakes and How to Fix Them](#).

Complete working examples

Dilution calculation -- Calculate the final concentration after diluting a stock solution, with randomized values and absolute tolerance.

```
## TITLE('Dilution calculation')
## DESCRIPTION
## Calculate the final concentration after diluting a stock
## solution to a larger final volume.
## ENDDescription
## KEYWORDS('dilution','concentration','numeric','tolerance')
## DBsubject('Biochemistry')
## DBchapter('Concentrations')
## DBsection('Dilution')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");
$stock = random(20, 80, 10);
$vol_stock = random(1, 4, 1);
$vol_final = random(10, 20, 5);
$final_conc = Real($stock * $vol_stock / $vol_final);

BEGIN_PGML
You dilute [$vol_stock] mL of a [$stock] mg/mL stock solution
to a final volume of [$vol_final] mL.

What is the final concentration (mg/mL)? [_____]{$final_conc}
END_PGML

ENDDOCUMENT();
```

Surface area to volume ratio -- Calculate a ratio using relative tolerance, demonstrating the `tolType` and `tolerance` options.

```
## TITLE('Surface area to volume ratio')
## DESCRIPTION
## Calculate the surface area to volume ratio of a spherical
## cell and report it with relative tolerance.
## ENDDescription
## KEYWORDS('ratio','relative tolerance','numeric')
## DBsubject('Cell Biology')
```

```
## DBchapter('Cell Size')
## DBsection('Surface Area')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");
$radius = random(2, 8, 1);
$ratio = Real(3 / $radius)->with(
    tolType => "relative",
    tolerance => 0.02,
);

BEGIN_PGML
A spherical cell has a radius of [$radius] micrometers.

The surface area to volume ratio of a sphere is  $3/r$ .

What is the surface area to volume ratio for this cell?
[_____]{$ratio}
END_PGML

ENDDOCUMENT();
```

Authoring notes

- State units in the prompt and use the same units in the expected answer.
- Choose tolerances that reflect rounding expectations (do not guess after the fact).

This page titled [5.5: Numerical Entry](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.6: Fill in the Blank

Fill-in-the-Blank (FIB) is best for short textual answers (terms, labels, short phrases) where you can list a small number of acceptable answers.

Text answers are a biology reality: names, labels, and short phrases show up everywhere. Decide in advance what you will accept for pluralization, hyphens, and capitalization, and list those variants explicitly. If spelling or naming is not the point, switch to recognition style (MC, matching) so the item grades the concept, not typing accuracy.

Inputs

- `question_text` (str)
- `answers_list` (list): acceptable answers

Science example (cell biology)

```
question_text:
    The organelle primarily responsible for ATP production in eukaryotic cells is the ____

answers_list:
    - mitochondrion
    - mitochondria
```

Official example (OpenWeBWorK PG samples): [StringsInContext](#).

Preferred PGML pattern

Use a string `Context` and `Compute` to define the expected answer, then attach it inline in the PGML blank.

```
Context("String");
$ans = Compute("mitochondrion");

BEGIN_PGML
The organelle primarily responsible for ATP production
in eukaryotic cells is the [_____]{$ans}.
END_PGML
```

If the string context is not available in your renderer snapshot, convert the question to multiple choice (`RadioButtons`) or a dropdown (`PopUp`) instead. This avoids grading failures from unsupported evaluator types. For answer blank syntax details, see [Answer blanks and answers](#). When a prompt requires multiple blanks, see [Multipart Questions](#) (Multi-Part Fill-in-the-Blank).

Complete working examples

Organelle naming -- Name the organelle responsible for ATP production using a String context answer.

```
## TITLE('Name the ATP-producing organelle')
## DESCRIPTION
## Name the organelle primarily responsible for ATP production
## in eukaryotic cells using a fill-in-the-blank format.
## ENDDescription
## KEYWORDS('organelles','fill in the blank','String')
## DBsubject('Cell Biology')
```

```
## DBchapter('Organelles')
## DBsection('Energy')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "contextArbitraryString.pl",
    "PGcourse.pl",
);

Context("ArbitraryString");
$ans = Compute("mitochondrion");

BEGIN_PGML
The organelle primarily responsible for ATP production
in eukaryotic cells is the [_____]{ $ans }.
END_PGML

ENDDOCUMENT();
```

Nucleic acid abbreviation -- Name the molecule that carries genetic information, demonstrating a simple string answer.

```
## TITLE('Name the genetic information molecule')
## DESCRIPTION
## Name the molecule that carries genetic information
## in most organisms.
## ENDDescription
## KEYWORDS('genetics','DNA','fill in the blank','String')
## DBsubject('Molecular Biology')
## DBchapter('Nucleic Acids')
## DBsection('DNA')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "contextArbitraryString.pl",
    "PGcourse.pl",
```

```
);  
  
Context("ArbitraryString");  
$ans = Compute("DNA");  
  
BEGIN_PGML  
The molecule that stores hereditary information in most  
living organisms is [_____]{ $ans }.  
END_PGML  
  
ENDDOCUMENT();
```

Authoring notes

- Decide whether you will accept plural vs singular forms, and list them explicitly.
- Keep capitalization rules explicit (either normalize answers or list variants).

This page titled [5.6: Fill in the Blank](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.7: Multipart Questions

Multi-Part Fill-in-the-Blank (MULTI_FIB) is best when students must supply several short answers within a single prompt. This matches many science workflows (labeling, short computed parts, or multi-step interpretation). For single-blank questions, see [Fill in the Blank](#) instead.

Multi-blank prompts are great for short workflows, but they fail when blanks are not clearly tied to a specific request. Keep each blank's request local to the blank, keep formatting consistent across blanks, and prefer two to three blanks unless the prompt is very structured. When students miss one part, they should still understand what each other blank was asking.

Inputs

- `question_text` (str)
- `answer_map` (dict): map from blank position to correct answer

Science example (genotype to phenotype)

```
question_text:
  For a monohybrid cross Aa x Aa:
  - The expected fraction of aa offspring is [blank_1].
  - The expected fraction of offspring showing the dominant phenotype is [blank_2].

answer_map:
  blank_1: 1/4
  blank_2: 3/4
```

Official example (OpenWeBWorK PG samples): [Multianswer](#).

Preferred PGML pattern

Define each answer as a separate named variable and attach each one inline in the PGML blank closest to its prompt (see [Answer blanks and answers](#) for answer blank syntax). Do not use anonymous evaluator arrays or rely on implicit `ANS( . . . )` ordering.

```
Context("Numeric");
$ans_a = Compute("1/4");
$ans_b = Compute("3/4");

BEGIN_PGML
For a monohybrid cross Aa x Aa:

Part A: The expected fraction of aa offspring is [_____]{ $ans_a }.

Part B: The expected fraction showing the dominant
phenotype is [_____]{ $ans_b }.
END_PGML
```

For mixed formats (for example a RadioButtons part and a numeric part), define each answer object separately and place its blank next to the corresponding text. Keep variable naming consistent (`$ans_a` , `$ans_b` , or `$ans1` , `$ans2`).

Complete working examples

Genetics cross fractions -- Calculate two expected fractions from a monohybrid cross between heterozygous parents.

```
## TITLE('Monohybrid cross fractions')
## DESCRIPTION
## Calculate the expected fraction of homozygous recessive
## and dominant phenotype offspring from Aa x Aa.
## ENDDescription
## KEYWORDS('genetics','monohybrid','multi-part','numeric')
## DBsubject('Genetics')
## DBchapter('Mendelian Genetics')
## DBsection('Monohybrid Cross')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");
$ans_a = Compute("1/4");
$ans_b = Compute("3/4");

BEGIN_PGML
For a monohybrid cross Aa x Aa:

Part A: The expected fraction of aa offspring is
[_____]{ $ans_a }.

Part B: The expected fraction showing the dominant
phenotype is [_____]{ $ans_b }.
END_PGML

ENDDOCUMENT();
```

Three-part dilution -- Calculate stock concentration, dilution factor, and final concentration across three answer blanks.

```
## TITLE('Three-part dilution problem')
## DESCRIPTION
## Given a dilution scenario, calculate the stock concentration
## in different units, the dilution factor, and the final
```

```
## concentration.
## ENDDescription
## KEYWORDS('dilution','multi-part','numeric','concentration')
## DBsubject('Biochemistry')
## DBchapter('Concentrations')
## DBsection('Dilution')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");
$stock_mM = random(10, 50, 10);
$stock_uM = Real($stock_mM * 1000);
$dilution_factor = list_random(5, 10, 20);
$final_uM = Real($stock_uM / $dilution_factor);

BEGIN_PGML
A stock solution has a concentration of [$stock_mM] mM.

Part A: What is the stock concentration in uM?
[_____][$stock_uM]

Part B: If you dilute it 1:[$dilution_factor], what is the
dilution factor? [_____][$dilution_factor]

Part C: What is the final concentration in uM?
[_____][$final_uM]
END_PGML

ENDDOCUMENT();
```

Authoring notes

- Keep blanks unambiguous and label them consistently.
- Use consistent formatting for answers (fractions vs decimals).

This page titled [5.7: Multipart Questions](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.8: Ordered List

Ordered List (ORDER) is best for sequencing steps in a process. In science courses it is useful for protocols and workflows (PCR, gel electrophoresis, sample prep), as long as the sequence is well-defined.

Ordered list questions work best for canonical cycles and short protocols, not for long procedures. If your course has multiple acceptable orders depending on reagent choice or instrument, this type will create disputes. Keep lists short and pick sequences students should be able to justify, not merely recite.

Inputs

- `question_text` (str)
- `ordered_answers_list` (list): correct sequence

Science example (PCR workflow)

```
question_text:
    Put the following PCR steps in the correct order.

ordered_answers_list:
    - Denaturation
    - Primer annealing
    - Extension
```

Official examples (OpenWeBWorK PG samples): [Problem techniques list](#).

Preferred PGML pattern (DraggableProof)

Use `draggableProof.pl` to create a drag-and-drop ordering widget (for general list formatting in PGML, see [Lists and workflows](#)). The inline `[_]{$ordering}` pattern is not supported in PG 2.17 (it produces an unrecognized evaluator error), so use the legacy `->Print` plus `ANS(...)` form instead.

```
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "draggableProof.pl",
    "PGcourse.pl",
);

$ordering = DraggableProof(
    [ "Denaturation", "Primer annealing", "Extension" ],
    [],
    SourceLabel => "Steps",
    TargetLabel => "Put in correct order",
);
$ordering_html = $ordering->Print;

BEGIN_PGML
Put the PCR steps in the correct order.
```

```
[$ordering_html]*  
END_PGML  
  
ANS($ordering->cmp);
```

Key points:

- The first array holds items in the correct order; the second array (usually empty) holds items that start in the target zone.
- Include `MathObjects.pl` ; it is required by the macro.
- Use `[$ordering_html]*` to inject the widget HTML without PGML escaping (the `*` flag is critical).
- Use `ANS($ordering->cmp)` for grading because inline attachment is not supported.

Complete working examples

PCR steps -- Order the three main steps of a PCR cycle using the DraggableProof drag-and-drop widget.

```
## TITLE('Order the PCR cycle steps')  
## DESCRIPTION  
## Put the three main steps of a PCR cycle in the correct  
## order using a drag-and-drop interface.  
## ENDDescription  
## KEYWORDS('PCR','ordering','DraggableProof')  
## DBsubject('Molecular Biology')  
## DBchapter('Techniques')  
## DBsection('PCR')  
## Date('2026-02-15')  
## Author('Author name')  
## Institution('Institution name')  
  
DOCUMENT();  
loadMacros(  
    "PGstandard.pl",  
    "MathObjects.pl",  
    "PGML.pl",  
    "draggableProof.pl",  
    "PGcourse.pl",  
);  
  
$ordering = DraggableProof(  
    [ "Denaturation", "Primer annealing", "Extension" ],  
    [],  
    SourceLabel => "Steps",  
    TargetLabel => "Put in correct order",  
);  
$ordering_html = $ordering->Print;  
  
BEGIN_PGML  
Put the three main steps of a PCR cycle in the correct order.
```

```
[$ordering_html]*  
END_PGML
```

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

```
ENDDOCUMENT();
```

Mitosis phases -- Order the four phases of mitosis in the correct sequence.

```
## TITLE('Order the phases of mitosis')  
## DESCRIPTION  
## Put the four phases of mitosis in the correct order  
## using a drag-and-drop interface.  
## ENDDescription  
## KEYWORDS('mitosis','cell division','ordering','DraggableProof')  
## DBsubject('Cell Biology')  
## DBchapter('Cell Division')  
## DBsection('Mitosis')  
## Date('2026-02-15')  
## Author('Author name')  
## Institution('Institution name')
```

```
DOCUMENT();  
loadMacros(  
    "PGstandard.pl",  
    "MathObjects.pl",  
    "PGML.pl",  
    "draggableProof.pl",  
    "PGcourse.pl",  
);  
  
$ordering = DraggableProof(  
    [ "Prophase", "Metaphase", "Anaphase", "Telophase" ],  
    [],  
    SourceLabel => "Phases",  
    TargetLabel => "Put in correct order",  
);  
$ordering_html = $ordering->Print;
```

```
BEGIN_PGML  
Put the four phases of mitosis in the correct order.
```

```
[$ordering_html]*  
END_PGML
```

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

```
ENDDOCUMENT( );
```

Authoring notes

- Only use ORDER when there is a single standard sequence for the context you are testing.
- Keep the list short (3-6 items) to avoid turning it into a memory game.

This page titled [5.8: Ordered List](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

5.9: Workflow and QA

This page combines a practical ADAPT workflow checklist with a compact QA pass for biology-style WeBWorK items. The goal is to give you a repeatable routine that turns "generic error" screens into specific fixes before you publish to students.

Treat previewing as a scientific replication step: same inputs should give the same result, and failures should be reproducible. For a systematic approach to testing and common pitfalls, see [Testing and Debugging](#). First pass should be fast and local to get line-level errors and sweep variants, then final pass should be in ADAPT to confirm student view and assignment behavior. If a platform only reports "error," that is a signal to change the testing tool, not to guess.

Workflow: clone, edit, preview, assign

1. Start from a working question (clone or copy a trusted template from the OPL, see [OPL Header and Metadata](#) for OPL metadata).
2. Edit one small thing at a time: story text, values, answer checker, or prompt structure.
3. Preview and revise until the student view and grading are stable.
4. Add the question to an assignment and confirm ordering and point values.
5. Publish, then spot-check student view for the assignment context.

QA: variants, blanks, text variants, units, plausibility

- **Variants:** preview multiple variants and watch for impossible biology and awkward rounding.
- **Blanks:** confirm each blank is bound to the intended answer checker and that blank order matches prompt order.
- **Text variants:** decide what you accept for plural/singular, hyphens, spacing, and case (gene and protein naming is a common trap).
- **Units:** prevent unit drift (μM vs mM, μL vs mL) and restate expected units next to the blank.
- **Plausibility:** constrain random ranges (no negative concentrations, no unrealistic pipette volumes).
- **Log-scale traps:** state the model and reporting format for pH, Ct, and fold change.

Recommended testing loop

- First pass: use local rendering to get line-level errors and to sweep variants quickly.
- Final pass: use ADAPT preview to confirm permissions, assignment behavior, and student view.

This page titled [5.9: Workflow and QA](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

6: Advanced PGML Techniques

This chapter introduces advanced PGML authoring patterns for richer layouts and interaction patterns. It covers CSS-based emphasis, tables with `niceTables.pl`, modern matching layouts, statement-by-statement evaluation, safer randomization habits, display graphs with `PGgraphmacros.pl`, and chemical structure rendering with `RDKit.js`.

Matching layouts include the one place where `MODES(HTML => ...)` is required: wrapper HTML built inside eval blocks for dynamic content.

These techniques assume the PG 2.17 subset used in ADAPT and webwork-pg-renderer, so some macros from full WeBWorK installs are not available. Use the patterns here when the basic PGML tools in Chapter 3 are not enough.

Start here (task first)

- I need visual emphasis or color in the prompt -> [Text Coloring and Emphasis](#).
- I need a data or layout table -> [Making Tables with niceTables](#).
- I need a two-column matching layout -> [Randomized Matching Problems](#).
- I need true/false per statement -> [Randomized MC True and False Statements](#).
- I need safer randomization -> [Advanced Randomization Techniques](#).
- I need a display graph (titration curve, data plot) -> [Making Graphs](#).
- I need to render chemical structures from SMILES -> [Rendering Chemical Structures with RDKit](#).
- I need to display chemical formulas or reaction equations -> [Displaying Chemical Formulas](#).

Quick patterns

Quick patterns: need, use, and page.

Need	Use	Page
Emphasize key words	HTML spans and PGML wrappers	Text Coloring and Emphasis
Structured data display	DataTable or LayoutTable	Making Tables with niceTables
Two-column matching	PopUp widgets + flexbox	Randomized Matching Problems
Evaluate multiple statements	RadioButtons per statement	Randomized MC True and False Statements
Stable random variants	problemSeed + guard checks	Advanced Randomization Techniques
Display graph (curve, data plot)	PGgraphmacros.pl + init_graph	Making Graphs
Chemical structure rendering	RDKit.js + SMILES strings	Rendering Chemical Structures with RDKit
Chemical formula / reaction display	mhchem via MathJax <code>\ce{ }</code>	Displaying Chemical Formulas

[6.1: Text Coloring and Emphasis](#)

[6.2: Making Tables with niceTables](#)

[6.3: Making Graphs](#)

[6.4: Advanced Randomization Techniques](#)

[6.5: Randomized Matching Problems](#)

[6.6: Randomized MC True and False Statements](#)

[6.7: Rendering Chemical Structures with RDKit](#)

[6.8: Displaying Chemical Formulas](#)

This page titled [6: Advanced PGML Techniques](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.1: Text Coloring and Emphasis

This page shows how to add visual emphasis using CSS-based styling that works in the PG 2.17 subset. Use this for terms like ABOVE/BELOW or NOT where color or bold weight prevents misreads.

Use this page when: you need to highlight key words, numeric values, or label classes in matching or statement-based prompts. For general PGML layout controls, see [Layout controls](#). For common CSS mistakes, see [Common Mistakes and How to Fix Them](#).

Why emphasize key information?

Students often miss critical details in question statements, leading to common errors:

- Reading the wrong value (pH 7.4 instead of pH 8.0)
- Missing important qualifiers like NOT, LEAST, or MOST
- Overlooking units or experimental conditions

Visual emphasis helps students quickly locate critical values and focus on what matters. Even a small amount of color or size change can dramatically reduce misreading errors.

Emphasis styles by use case

Emphasis styles by use case: use case, style, css properties, and more.

Use case	Style	CSS properties	Example content
Numeric values (pH, temperatures)	Blue badge	color:#0066cc; font-size:1.35em; font-weight:700; background-color:#e6f2ff; padding:0.1em 0.4em; border-radius:4px	pH 7.4
Key terms (hydrophobic, polar)	Gold emphasis	color:#997300; font-size:1.25em; font-weight:700	hydrophobic
Negation (NOT, EXCEPT)	Red warning	color:#cc0000; font-size:1.3em; font-weight:700	NOT
Multiple values (pKa lists)	Monospace with background	font-family:monospace; background-color:#f5f5f5; padding:0.2em 0.5em; border-radius:3px; font-size:1.1em	1.82, 6.00, 9.17

CSS class approach (preferred)

Define CSS classes once in `HEADER_TEXT` at the top of the problem, then reference them with PGML tag wrappers. This keeps styles in one place and reusable throughout the problem.

```

HEADER_TEXT(MODES(TeX => '', HTML => <<'END_STYLE'));
<style>
.emph-data      { color:#0066cc; font-size:1.35em; font-weight:700;
                  background-color:#e6f2ff; padding:0.1em 0.4em; border-radius:4px; }
.emph-term      { color:#997300; font-size:1.25em; font-weight:700; }
.emph-warning   { color:#cc0000; font-size:1.3em; font-weight:700; }
</style>
END_STYLE

BEGIN_PGML
Which amino acid is predominantly charged at
[<pH 7.4>][['span', class => 'emph-data']][['', '']]?

```

```
Which of the following is
[<NOT>][{'span', class => 'emph-warning'}][{'',''}]
a nucleotide?
END_PGML
```

The tag wrapper syntax is `[<text>][{'span', class => 'class-name'}][{'',''}]`. The first brace pair sets the HTML element and attributes; the second `{['','']}` closes the tag.

Inline style approach

If you only need emphasis in one place, you can use inline styles directly in the tag wrapper. This works but is harder to maintain when the same style appears in multiple places.

```
BEGIN_PGML
[<Important>][{'span', style => 'color:#b91c1c; font-weight:700;'}][{'',''}]
END_PGML
```

What does NOT work: TeX and MathJax color commands

Do not use `\color`, `\textcolor`, or other TeX/MathJax color commands for text coloring in PGML. In this install, PGML escapes backslashes into `tex2jax_ignore` spans before MathJax processes them, so the TeX color commands never reach the math renderer.

```
## These all FAIL - do not use them ##
[\color{#cc00cc}{\text{colored text}}]
\(\color{#cc00cc}{\text{colored text}}\)
$colored = "\color{red}{\text{warning}}";
[$colored]
```

Backslashes are escaped into `<span class="tex2jax_ignore">\</span>` before MathJax runs, so the TeX commands are never processed. Use CSS styling instead.

Passing HTML variables through PGML

PGML tag wrappers are parsed at compile time. If you build HTML spans inside Perl variables, PGML will not re-parse them. Instead, pass them through using `[$var]*` so the HTML is not escaped.

```
# Build the HTML span in the Setup section
$label = "<span style='color:#0f766e; font-weight:700;'>ABOVE</span>";

BEGIN_PGML
Threshold: [$label]*
END_PGML
```

The trailing `*` tells PGML to pass the content through as raw HTML. Without it, the HTML tags are escaped and displayed as literal text.

Font size guidelines

Font size guidelines: size, level of emphasis, and when to use.

Size	Level of emphasis	When to use
1.1em	Subtle	Formulas, value lists, monospace data
1.25em	Moderate	Key terms, qualitative descriptors
1.35em	Strong	pH values, critical numeric data
1.5em	Maximum	Rarely needed; only for prominent warnings

Do not exceed 1.5em. Larger text becomes distracting and looks unprofessional.

Color palette recommendations

Color palette recommendations: color, hex value, and use case.

Color	Hex value	Use case
Blue	#0066cc	pH values, temperatures, concentrations
Gold	#997300	Key terms like hydrophobic, polar, most, least
Red	#cc0000	NOT, EXCEPT, CANNOT (use sparingly)
Light blue bg	#e6f2ff	Data badge backgrounds behind blue text
Light gray bg	#f5f5f5	Subtle monospace backgrounds for value lists

Avoid pure black (#000000) for emphasis, bright green or yellow (accessibility issues), and using more than two or three colors per question (visual clutter).

Accessibility considerations

Never rely on color alone to convey meaning. Combine color with at least one other method such as font size, font weight, or a background highlight so that colorblind users can still distinguish emphasized content. Aim for a minimum contrast ratio of 4.5:1 between text and background. The recommended palette above meets this threshold on white backgrounds (#0066cc = 8.6:1, #997300 = 5.2:1, #cc0000 = 5.9:1).

Complete working examples

pH question with blue badge (CSS class approach):

```

HEADER_TEXT(MODES(TeX => '', HTML => <<'END_STYLE'));
<style>
.emph-ph { color:#0066cc; font-size:1.35em; font-weight:700;
           background-color:#e6f2ff; padding:0.1em 0.4em; border-radius:4px; }
</style>
END_STYLE

$ph_value = list_random(6.0, 7.0, 7.4, 8.0);

BEGIN_PGML
Which amino acid is predominantly charged at
[<pH [$ph_value]>][['span', class => 'emph-ph']]{'', ''}]?

```

```
[_]{ $answer }  
END_PGML
```

Negation question with red warning (variable approach):

```
$not_emph = "<span style='color:#cc0000; font-size:1.3em; font-weight:700;'>NOT</span>  
  
BEGIN_PGML  
Which of the following is [$not_emph]* a nucleotide?  
  
[_]{ $answer }  
END_PGML
```

Multiple emphasized elements (mixed data and term emphasis):

```
HEADER_TEXT(MODES(TeX => '', HTML => <<'END_STYLE'));  
<style>  
.emph-data { color:#0066cc; font-size:1.25em; font-weight:700;  
             background-color:#e6f2ff; padding:0.1em 0.3em; border-radius:3px; }  
.emph-term { color:#997300; font-size:1.25em; font-weight:700; }  
</style>  
END_STYLE  
  
$compound = "glucose";  
$temp = "37 degrees C";  
  
BEGIN_PGML  
At [<[$temp]>][['span', class => 'emph-data']][['', '']],  
what is the solubility of  
[<[$compound]>][['span', class => 'emph-term']][['', '']]  
in water?  
  
[_]{ $answer } g/L  
END_PGML
```

Complete renderable problem with emphasis -- A full problem using CSS class emphasis to highlight a pH value in the question stem.

```
## TITLE('Amino acid charge at a given pH')  
## DESCRIPTION  
## Identify whether an amino acid side chain is charged  
## at a specified pH, with the pH value visually emphasized.  
## ENDDescription  
## KEYWORDS('amino acids', 'pH', 'emphasis', 'RadioButtons')  
## DBsubject('Biochemistry')  
## DBchapter('Amino Acids')  
## DBsection('Charge')
```

```
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

HEADER_TEXT(MODES(TeX => '', HTML => <<'END_STYLE'));
<style>
.emph-ph { color:#0066cc; font-size:1.35em; font-weight:700;
          background-color:#e6f2ff; padding:0.1em 0.4em;
          border-radius:4px; }
</style>
END_STYLE

$rb = RadioButtons(
    [ 'Positively charged', 'Negatively charged',
      'Neutral', 'Zwitterionic only' ],
    'Positively charged',
    labels      => 'ABC',
    displayLabels => 1,
);

BEGIN_PGML
The side chain of lysine has a pKa of approximately 10.5.

At [<pH 7.4>][{'span', class => 'emph-ph'}][['', '']],
is the lysine side chain:

[_]{$rb}
END_PGML

ENDDOCUMENT();
```

Matching problem label colors

When labels in matching problems need fixed colors by meaning (not by randomized position), build an HTML mapping in Perl and emit each label with `*` so PGML does not escape the spans. This keeps colors tied to the label text even when the display order is randomized.

```
%answer_html = (
    'ionic bond'      => "<span style='color:#009900; font-weight:700;'>ionic bond</span>"
```

```
'hydrogen bond' => "<span style='color:#e60000; font-weight:700;'>hydrogen bond</span>";
);
@answers_html = map { $answer_html{$_} || $_ } @answers;

# In PGML, render each label with * to pass through HTML
[$answers_html[$shuffle[0]]]*
```

Common failure: missing the asterisk

```
[$label]      # BROKEN - HTML is escaped, shows raw tags
[$label]*     # CORRECT - asterisk prevents escaping
```

If you see literal `<span>` tags in the rendered question, the most likely cause is a missing `*`. Always use `[$var]*` when the variable contains HTML.

Best practices

Do

- Emphasize one to three key elements per question.
- Use consistent colors for similar element types across your problem set.
- Combine color with size or weight for accessibility.
- Create emphasis variables in the Setup section, not inside PGML.
- Use `[$var]*` to render HTML stored in Perl variables.
- Prefer CSS classes defined in HEADER_TEXT over inline styles.

Do not

- Emphasize everything; it defeats the purpose.
- Use more than three colors in one question.
- Make text larger than 1.5em.
- Use red for non-warning content.
- Use TeX or MathJax color commands; they do not render in this install.
- Forget the `*` in `[$var]*` when the variable contains HTML.

This page titled [6.1: Text Coloring and Emphasis](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.2: Making Tables with niceTables

This page explains how to build tables with `niceTables.pl` (see [Common PG Macros](#) for macro loading), which is the only supported way to create tables in this install. For basic table concepts in PGML, see [Tables and structured data](#). Standard HTML table tags (`<table>` , `<tr>` , `<td>` , `<th>`) are blocked by the server and will not render. Use `DataTable` or `LayoutTable` instead.

Use this page when: you need to present measurements, observations, or structured data in a readable format inside a WeBWorK problem.

Why niceTables is the only option

The WeBWorK PG rendering engine strips raw HTML table tags for security reasons. If you write `<table><tr><td>...</td></tr></table>` inside a problem, the tags are silently removed and the cell contents collapse into a single line. The `niceTables.pl` macro generates safe, styled tables that the server allows through. Every table in every problem must use this macro.

DataTable vs LayoutTable

DataTable vs LayoutTable: table type, use it for, and notes.

Table type	Use it for	Notes
DataTable	Real data with headers, captions, or semantic meaning (measurements, conditions, experimental results).	Adds header structure and improves accessibility. Supports the <code>caption</code> option.
LayoutTable	Visual layout: side-by-side answer blanks, aligned labels, widget grids.	No header semantics. Use sparingly; do not fake data tables with LayoutTable.

Rule of thumb: if the content has a header row or a caption, use `DataTable` . If you are arranging blanks or labels spatially, use `LayoutTable` .

Copy and edit (DataTable)

This template shows a three-row DataTable with a header row. Customize the row data, column headers, and options to match your experiment.

```
loadMacros(
  "PGstandard.pl",
  "MathObjects.pl",
  "PGML.pl",
  "niceTables.pl",
);

$rows = [
  ['Tube', 'Stock (mM)', 'Final (uM)'],
  ['A', 2.0, ''],
  ['B', 4.0, ''],
  ['C', 8.0, ''],
];

$tbl = DataTable($rows, columns => 'left', center => 1, horizontalrules => 1);
```

```
BEGIN_PGML
[@ $tbl @]*
END_PGML
```

Copy and edit (LayoutTable)

Use LayoutTable when you need to arrange answer blanks or labels side by side without header semantics. Customize the cell contents and answer variables.

```
$layout = LayoutTable(
  [
    [ 'Lane 1', 'Lane 2', 'Lane 3' ],
    [ '[_]{ $ans1 }', '[_]{ $ans2 }', '[_]{ $ans3 }' ],
  ],
  center => 1,
  horizontalrules => 1,
);

BEGIN_PGML
[@ $layout @]*
END_PGML
```

Cell content formats

Most cells are plain text strings. Use a hashref or arrayref only when you need per-cell alignment or CSS overrides.

Each cell in the 2-D array you pass to `DataTable` or `LayoutTable` can be one of three shapes.

- **Plain text** ('hello') -- most cells. Just pass a string or number.
- **Hashref** ({ data => 'hello', halign => 'c' }) -- when you need per-cell options like alignment or CSS.
- **Arrayref** (['hello', halign => 'c']) -- shorthand for hashref. Item 0 is content; the rest are key/value options.

Table-level options reference

These options are passed as key/value pairs after the row data.

- `center` -- set to 1 to center the table horizontally.
- `caption` -- add a caption above the table (DataTable only).
- `horizontalrules` -- set to 1 to draw horizontal lines between rows.
- `valign` -- vertical alignment: `top`, `middle`, or `bottom`.
- `padding` -- array [vertical, horizontal] specifying cell padding in em.

CSS hooks for fine-grained styling:

- `tablecss` -- CSS applied to the outer table element.
- `datacss` -- CSS applied to every data cell.
- `headercss` -- CSS applied to every header cell.
- `allcellcss` -- CSS applied to every cell (header and data).
- `columnscss` -- arrayref of CSS strings, one per column.

PGML table syntax

PGML's inline table syntax is convenient for simple tables that do not need heavy customization.

PGML also supports an inline table syntax using `[# ... #]`. This is backed by `niceTables` internally and is acceptable when it improves readability. Each row uses `[. ... .]` for cells.

```
BEGIN_PGML
[# 'Tube' , 'Stock (mM)' , 'Final (uM)' #]
[. 'A' , 2.0 , [_____]{$ans1} .]
[. 'B' , 4.0 , [_____]{$ans2} .]
[. 'C' , 8.0 , [_____]{$ans3} .]
END_PGML
```

For complex tables with many options, the explicit `DataTable()` or `LayoutTable()` call gives you more control.

Translating HTML tables

If you are converting a problem from another platform or an older PG format, you will likely encounter raw HTML tables that need translation.

If you are converting a problem that originally used raw HTML tables, follow these rules.

- **Header row or caption present:** use `DataTable()`. Map `<th>` cells to the first row of the array.
- **Pure layout (two-column lists, widget grids):** use `LayoutTable()`.
- **Nested tables:** not supported. Flatten the content into a single table or use separate tables.
- **rowspan or colspan:** not supported. Restructure the data into a simple rectangular grid.
- Never emit raw HTML table tags as a fallback. If the table cannot be translated, rewrite it manually.

Biology examples

Dilution series

A `DataTable` showing stock concentrations where students compute the final concentration.

```
$dilution = DataTable(
  [
    ['Tube', 'Stock (mM)', 'Dilution', 'Final (uM)'],
    ['A', 2.0, '1:10', ''],
    ['B', 4.0, '1:10', ''],
    ['C', 8.0, '1:10', ''],
  ],
  caption => 'Serial dilution worksheet',
  center => 1,
  horizontalrules => 1,
);
```

Enzyme kinetics

Substrate concentration vs reaction velocity with one computed blank.

```
$kinetics = DataTable(
  [
    ['[S] (mM)', 'Velocity (umol/min)'],
    [0.1, 12],
    [0.5, 45],
    [1.0, '[_____]{$vmax_half}'],
  ],
```

```
[5.0, 88],
],
center => 1,
horizontalrules => 1,
);
```

Gel or blot comparison

A LayoutTable that arranges lane labels above answer blanks so students identify bands.

```
$gel = LayoutTable(
[
[ 'Marker', 'Sample 1', 'Sample 2', 'Control' ],
[ '[_{ $band1 }]', '[_{ $band2 }]',
'[_{ $band3 }]', '[_{ $band4 }]' ],
],
center => 1,
allcellcss => 'padding: 4px 8px;',
);
```

Complete working example

Enzyme kinetics data table -- A full problem presenting substrate concentration vs velocity data in a DataTable, with a RadioButtons question about the data pattern.

```
## TITLE('Enzyme kinetics data interpretation')
## DESCRIPTION
## Review an enzyme kinetics data table and identify the
## relationship between substrate concentration and velocity.
## ENDDescription
## KEYWORDS('enzyme kinetics','DataTable','niceTables','RadioButtons')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Kinetics')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "niceTables.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);
```

```
$tbl = DataTable(  
  [  
    ['[S] (mM)', 'Velocity (umol/min)'],  
    [0.1, 12],  
    [0.5, 45],  
    [1.0, 68],  
    [5.0, 88],  
    [10.0, 95],  
  ],  
  center => 1,  
  horizontalrules => 1,  
);  
  
$rb = RadioButtons(  
  [  
    'Velocity increases linearly with [S]',  
    'Velocity approaches a maximum as [S] increases',  
    'Velocity decreases as [S] increases',  
    'Velocity is independent of [S]' ],  
  'Velocity approaches a maximum as [S] increases',  
  labels      => 'ABC',  
  displayLabels => 1,  
);  
  
BEGIN_PGML  
The following table shows enzyme kinetics data.  
  
[@ $tbl @]*  
  
Which statement best describes the relationship between  
substrate concentration and reaction velocity?  
  
[_]{$rb}  
END_PGML  
  
ENDDOCUMENT();
```

Apply it today

- Use `DataTable` for real measurements and `LayoutTable` only for visual arrangement.
- Never use raw HTML table tags; they are blocked and will not render.
- Start with the copy-and-edit examples above, then customize options as needed.

This page titled [6.2: Making Tables with niceTables](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.3: Making Graphs

This page covers how to create graphs in WeBWorK PG/PGML problems. It focuses on `PGgraphmacros.pl`, which produces static GD bitmap images suitable for display-only graphs such as titration curves, enzyme kinetics plots, and dose-response data.

Use this page when: you need to show a graph in a problem but students answer with RadioButtons, PopUp menus, or numeric blanks rather than by clicking on the graph itself. Load `PGgraphmacros.pl` via the macro block (see [Common PG Macros](#)), set up graph variables in the Setup block (see [Setup](#)), and always test the rendered output (see [Testing and Debugging](#)).

Which graph macro to use

Which graph macro to use: need, macro, and notes.

Need	Macro	Notes
Display-only graph (titration curve, data plot)	<code>PGgraphmacros.pl</code>	GD bitmap image, no answer entry
Student clicks or draws on graph	<code>parserGraphTool.pl</code>	JS canvas, registers an answer entry

Rule of thumb: if students answer via RadioButtons or other PGML widgets (not by clicking the graph), use `PGgraphmacros.pl`. GraphTool creates a phantom answer entry that can interfere with subsequent PGML answer evaluators.

Minimal example with PGgraphmacros

Copy this block into a new problem file, save, and preview. It draws a single sine curve on a labeled grid.

```
loadMacros(
  'PGstandard.pl',
  'MathObjects.pl',
  'PGML.pl',
  'PGgraphmacros.pl',
  'PGcourse.pl',
);

# Create graph: init_graph(xmin, ymin, xmax, ymax, ...)
$gr = init_graph(-1, 0, 10, 14,
  axes => [0, 0],
  grid => [11, 14],
  size => [480, 400],
);

# Plot a function (string formula, not a code reference)
add_functions($gr,
  "2 + 3*sin(x) for x in <0,10> using color:blue and weight:2"
);

# Render as an image
$graph_img = image(insertGraph($gr), width => 480, height => 400, tex_size => 700);

BEGIN_PGML
Here is a graph:
```

```
[$graph_img]*  
END_PGML
```

Key points:

- `init_graph` returns a GD graph object.
- `add_functions` takes a **string** formula with `for x in <min,max> using color:COLOR and weight:N`.
- `image(insertGraph($gr), ...)` produces an HTML img tag.
- Use `[$graph_img]*` in PGML. The asterisk prevents HTML escaping.

init_graph parameters

```
$gr = init_graph($xmin, $ymin, $xmax, $ymax,  
  axes => [$x_origin, $y_origin], # where axes cross  
  grid => [$x_divisions, $y_divisions], # grid line count  
  size => [$width_px, $height_px], # image dimensions  
);
```

- `axes => [0, 0]` places the origin at (0, 0).
- `grid` controls the number of light grid lines (not tick labels).
- `size` sets the GD image dimensions in pixels.

Adding labels

Labels are positioned in graph coordinates (not pixel coordinates). The constructor is:

```
new Label($x, $y, $text, $color, $h_align, $v_align)
```

Horizontal alignment: 'left', 'center', 'right'. Vertical alignment: 'top', 'middle', 'bottom'.

Axis labels at edges of the plot area

```
$gr->lb(new Label($xmax, $ymin - 0.7, 'x-axis label', 'black', 'right', 'top'));  
$gr->lb(new Label($xmin - 0.1, $ymax, 'y-axis', 'black', 'right', 'top'));
```

Tick labels

```
# Tick labels on y-axis  
for $yv (0, 2, 4, 6, 8, 10, 12) {  
  $gr->lb(new Label($xmin - 0.08, $yv, $yv, 'black', 'right', 'middle'));  
}  
  
# Tick labels on x-axis  
for $xv (0, 1, 2, 3) {  
  $gr->lb(new Label($xv, $ymin - 0.3, $xv, 'black', 'center', 'top'));  
}
```

add_functions syntax and computed coefficients

The function string format is:

```
"FORMULA for x in <XMIN,XMAX> using color:COLOR and weight:WEIGHT"
```

- FORMULA : a Perl math expression using x as the variable (e.g., $2*x**3 + x - 1$).
- <XMIN,XMAX> : domain interval (angle brackets).
- color:COLOR : blue , red , green , black , etc.
- weight:WEIGHT : line thickness in pixels (1, 2, 3, ...).

For computed coefficients, interpolate Perl variables into the string:

```
$a = 0.5;
$b = -1.2;
$c = 3.0;
$d = 1.5;
$func_str = "$a*x**3 + $b*x**2 + $c*x + $d";
add_functions($gr, "$func_str for x in <0,3> using color:blue and weight:2");
```

Multiple curves

Pass multiple formula strings to `add_functions` . Each string specifies its own color and weight.

```
add_functions($gr,
  "$curve1 for x in <0,10> using color:blue and weight:2",
  "$curve2 for x in <0,10> using color:red and weight:2",
);
```

Point-by-point curves with moveTo and lineTo

When a formula string cannot express the curve shape (parametric curves, physical models where x is not the independent variable), draw point-by-point with `moveTo` and `lineTo` . Use 300 to 400 points for smooth results.

```
$first_pt = 1;
for $i (0..400) {
  $t = $i / 400.0;
  # compute ($x_val, $y_val) from $t
  if ($first_pt) {
    $gr->moveTo($x_val, $y_val);
    $first_pt = 0;
  } else {
    $gr->lineTo($x_val, $y_val, 'blue', 2);
  }
}
```

This approach is essential when the x-axis value is a computed function of the sweep variable rather than the independent variable itself.

Dashed lines

PGgraphmacros has no built-in dash style. Draw dashes manually as short line segments with gaps:

```
# Horizontal dashed line at y=$pka from x=0 to x=$xeq
for $d (0..29) {
```

```
$x1d = $d * 0.10;
$x2d = $x1d + 0.05;
if ($x1d < $xeq) {
  if ($x2d > $xeq) { $x2d = $xeq; }
  $gr->moveTo($x1d, $pka);
  $gr->lineTo($x2d, $pka, 'gray', 1);
}
}
```

Tune dash length (0.05) and gap (0.05) relative to graph coordinates. For vertical dashes, swap x and y in the loop.

Dots and stamps

Use `stamps` with `closed_circle` or `open_circle` to mark individual points:

```
# Filled circle at a point
$gr->stamps(closed_circle($x, $y, 'black'));

# Open circle (hollow)
$gr->stamps(open_circle($x, $y, 'black'));
```

Label sizing: why labels look small

Labels in PGgraphmacros are GD bitmap text. This has important consequences:

- `tex_size` on `image()` only affects TeX/PDF output, not HTML/GD output.
- TeX size commands (`\Large` , `\huge`) in label strings render as literal text in GD, not as formatting.
- `GD::Font->Giant` is the largest GD bitmap font (9 by 15 pixels) and is still small.
- **Rendering at 2x and displaying at 1x makes labels half-size.** GD text is fixed-pixel; there is no display-DPI concept.

GD font sizes

Available via `$label->font(GD::Font->NAME)` :

GD font sizes: font, pixel size, and notes.

Font	Pixel size	Notes
Tiny	5 x 8	Too small for most uses
Small	6 x 12	Default in most PG builds
MediumBold	7 x 13	Slightly larger
Large	8 x 16	Good default choice
Giant	9 x 15	Largest available

All are fixed-size bitmap fonts. The size difference between Small and Giant is modest.

Pragmatic fixes for readable labels

Keep `size` at the final display dimensions (no 2x trick). Improve readability through layout:

```
# Slightly larger canvas with extra margin for labels
$gr = init_graph(-0.6, -1.2, 3.4, 12.5,
  axes => [0, 0],
```

```

    grid => [8, 5],
    size => [520, 420],
  );

  # Axis labels at edges (right-aligned at plot boundary)
  $gr->lb(new Label(3.4, -0.7, 'OH- (equivalents)', 'black', 'right', 'top'));
  $gr->lb(new Label(-0.1, 12.5, 'pH', 'black', 'right', 'top'));

  # Fewer tick labels to reduce crowding
  for $yv (0, 2, 4, 6, 8, 10, 12) {
    $gr->lb(new Label(-0.08, $yv, $yv, 'black', 'right', 'middle'));
  }

  $graph_img = image(insertGraph($gr), width => 520, height => 420, tex_size => 700);

```

Key techniques:

- **Widen graph bounds** beyond data range (-0.6 to 3.4 instead of 0 to 3) to create margin space for labels.
- **Place axis labels at edges** using `$xmax` / `$ymax` coordinates with 'right', 'top' alignment.
- **Reduce tick label count** to avoid crowding (every 2 or 4 units instead of every 1).
- **Use high contrast** (black text on white background).

Worked example: triprotic titration curve

A single cubic polynomial **cannot** produce the correct titration curve shape for a molecule with three pKa values. A cubic has at most one inflection point, but a 3-pKa titration needs three sigmoid steps. The correct approach uses a physical speciation model.

Why not a cubic?

A cubic $ax^3 + bx^2 + cx + d$ through four points:

- Has only **one inflection point** -- a triprotic titration needs three sigmoid transitions.
- Anchors pKa values at integer equivalents (1, 2) instead of the correct half-equivalents (0.5, 1.5, 2.5).
- Produces a smooth bend, not the characteristic staircase-with-shoulders shape.

Speciation model (correct shape)

Sweep pH and compute the equivalents of base (n_{bar}) from the charge balance:

$$n_{\text{bar}} = \frac{(K_{a1}H^2 + 2K_{a1}K_{a2}H + 3K_{a1}K_{a2}K_{a3})}{(H^3 + K_{a1}H^2 + K_{a1}K_{a2}H + K_{a1}K_{a2}K_{a3})}$$

This gives three sigmoid steps with buffer flats at half-equivalence points (0.5, 1.5, 2.5 equivalents) and steep transitions at equivalence points (1.0, 2.0 equivalents).

Copy and edit (triprotic titration)

```

$Ka1_val = 10**(-$pKa1_num);
$Ka2_val = 10**(-$pKaR_num);
$Ka3_val = 10**(-$pKa2_num);

$num_pts = 400;
$ph_lo = 0.5;

```

```

$ph_hi = 12.5;
$first_pt = 1;
for $i (0..$num_pts) {
    $ph_sweep = $ph_lo + ($ph_hi - $ph_lo) * $i / $num_pts;
    $H_conc = 10**(-$ph_sweep);

    $denom = $H_conc**3
        + $Ka1_val * $H_conc**2
        + $Ka1_val * $Ka2_val * $H_conc
        + $Ka1_val * $Ka2_val * $Ka3_val;
    $nbar = ($Ka1_val * $H_conc**2
        + 2 * $Ka1_val * $Ka2_val * $H_conc
        + 3 * $Ka1_val * $Ka2_val * $Ka3_val) / $denom;

    if ($nbar >= 0 && $nbar <= 3.0) {
        if ($first_pt) {
            $gr->moveTo($nbar, $ph_sweep);
            $first_pt = 0;
        } else {
            $gr->lineTo($nbar, $ph_sweep, 'blue', 2);
        }
    }
}

```

This sweeps pH (y-axis) uniformly and computes equivalents (x-axis) as the dependent variable. The `moveTo` / `lineTo` approach handles this naturally since x is not the sweep variable.

Alternative: stacked logistic sigmoids

If you do not need physical accuracy, three logistic functions centered at half-equivalents approximate the shape. Note that this version does **not** pass exactly through pKa at each half-equivalence point; the speciation model is preferred for accuracy.

```

$k = 10; # steepness
$y0 = 1.0;
$y_top = 11.5;

for $i (0..300) {
    $xx = 3.0 * $i / 300.0;
    $s1 = 1 / (1 + exp(-$k * ($xx - 0.5)));
    $s2 = 1 / (1 + exp(-$k * ($xx - 1.5)));
    $s3 = 1 / (1 + exp(-$k * ($xx - 2.5)));

    $ph = $y0
        + ($pKa1_num - $y0) * $s1
        + ($pKaR_num - $pKa1_num) * $s2
        + ($pKa2_num - $pKaR_num) * $s3
        + ($y_top - $pKa2_num) * $s3 * 0.20;
}

```

```
# draw with moveTo/lineTo as shown above
}
```

GraphTool warning: answer evaluator interference

`parserGraphTool.pl` creates a JS-based interactive canvas where students can draw or place points. Its `ans_rule()` registers an answer entry (typically `AnSwEr0001`). This shifts the numbering of subsequent PGML answer blanks and can cause "Unrecognized evaluator type ||" errors on later RadioButtons or other PGML widgets.

Workaround (if you must use GraphTool alongside PGML answers):

- Place GraphTool in a `BEGIN_TEXT` block with `\{$gt->ans_rule()\}`.
- Use `LABELED_ANS($gt_ans_id, $gt_checker)` before the PGML block.
- Keep RadioButtons and other widgets in a separate `BEGIN_PGML` block.

Better approach: if the graph is display-only, use `PGgraphmacros.pl` instead.

Sizing and margins

The `size` parameter in `init_graph` sets the GD image pixel dimensions. The `width` and `height` in `image()` control the HTML display size. **Always keep these matched at 1x** -- rendering at 2x and displaying at 1x shrinks bitmap labels.

```
# Good: matched sizes, extra margin from wider bounds
$gr = init_graph(-0.6, -1.2, 3.4, 12.5,
  axes => [0, 0],
  grid => [8, 5],
  size => [520, 420],
);
$graph_img = image(insertGraph($gr), width => 520, height => 420, tex_size => 700);
```

To create margin space for labels without shrinking the data region, widen the graph bounds beyond the data range (e.g., `-0.6` to `3.4` for data spanning 0 to 3).

`tex_size` controls the image width in TeX/PDF output only (700 = 70% of text width). It has no effect on HTML/GD rendering.

Variable scoping reminder

Do **not** use `my` on any variable that PGML needs to interpolate. PG runs in a safe compartment where `my` variables are invisible to PGML.

```
# WRONG - invisible to PGML
my $graph_img = image(insertGraph($gr), ...);

# RIGHT - package-level, visible to PGML
$graph_img = image(insertGraph($gr), ...);
```

Unavailable macros

The following graph-related macros exist in the OPL but are **not** available in the local PG 2.17 renderer:

- `PGlateximage.pl` (LaTeX-to-image pipeline, requires LaTeX installation)
- `PGtikz.pl` (TikZ/pgfplots, requires LaTeX installation)

Stick to `PGgraphmacros.pl` for display graphs and `parserGraphTool.pl` for interactive graphs.

Complete working example

Simple function graph with numeric answer -- A full problem that plots a linear function and asks students to read a value from the graph.

```
## TITLE('Read a value from a graph')
## DESCRIPTION
## A linear function is plotted on a graph. Students read
## the y-value at a given x-coordinate.
## ENDDescription
## KEYWORDS('graph','PGgraphmacros','numeric','function plot')
## DBsubject('Biostatistics')
## DBchapter('Data Visualization')
## DBsection('Reading Graphs')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGgraphmacros.pl",
    "PGcourse.pl",
);

Context("Numeric");
$m = random(1, 3, 1);
$b = random(1, 4, 1);
$x_ask = random(2, 5, 1);
$y_ans = Real($m * $x_ask + $b);

$gr = init_graph(-1, -1, 8, 20,
    axes => [0, 0],
    grid => [9, 21],
    size => [400, 400],
);
add_functions($gr,
    "$m*x + $b for x in <0,8> using color:blue and weight:2"
);

$graph_img = image(insertGraph($gr),
    width => 400, height => 400, tex_size => 700);

BEGIN_PGML
The graph below shows a linear function.
```

```
[$graph_img]*
```

```
What is the y-value when x = [$x_ask]? [_____]{$y_ans}
```

```
END_PGML
```

```
ENDDOCUMENT();
```

Apply it today

- Use `PGgraphmacros.pl` for any display-only graph. Reserve `parserGraphTool.pl` for problems where students answer by drawing.
- Match `init_graph size to image() width/height` at 1x so bitmap labels stay readable.
- Widen graph bounds to create margin space for axis labels and tick marks.
- Use `moveTo` / `lineTo` for curves where x is not the independent variable (titration curves, parametric shapes).
- Avoid `my` on any variable referenced inside PGML blocks.

This page titled [6.3: Making Graphs](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.4: Advanced Randomization Techniques

This page summarizes safer randomization patterns in the PG 2.17 subset. The goal is to avoid impossible values, repeated variants, and hard-to-reproduce failures.

Use this page when: you need multi-step randomization or constraints between values, or you want to choose the right randomization function for your situation. Randomization variables are typically defined in the Setup block (see [Setup](#)). For testing randomized variants, see [Testing Randomization and Edge Cases](#).

Built-in random functions are already seeded

The PG environment automatically seeds the random number generator with `$problemSeed` before your problem code runs. For most problems, you can use the built-in random functions directly without any manual seeding. The same `$problemSeed` always produces the same sequence of random values, so student results are reproducible.

```
# These functions are already deterministic based on $problemSeed.
# No manual seeding required for most problems.
$value = random(1, 10, 1);
$item = list_random(@items);
$nonzero = non_zero_random(-5, 5, 1);
```

Core randomization functions (PGbasicmacros.pl)

Core randomization functions (PGbasicmacros.pl) reference.

Function	What it does	When to use it
<code>random(begin, end, incr)</code>	Returns a random number from begin to end stepping by incr.	Numeric ranges like concentrations, volumes, or counts.
<code>non_zero_random(begin, end, incr)</code>	Same as random but excludes zero from the result.	Denominators, divisors, or any value that must not be zero.
<code>list_random(@list)</code>	Returns one randomly chosen element from the list.	Choosing from named items like enzymes, organisms, or codons.
<code>SRAND(seed)</code>	Resets the global PG random generator to a specific seed.	Rarely needed. The environment already seeds with problemSeed.

Other randomization helpers

- `random_coprime([arrays...])` and `random_pairwise_coprime([arrays...])` from `PGauxiliaryFunctions.pl` return coprime integers.
- `randomOrder(...)` in `parserPopUp.pl` and `parserRadioButtons.pl` shuffles answer choices.
- `randomPrime(start, end)` from `contextInteger.pl` returns a random prime in the given range.

Statistics distribution functions

These functions generate random samples from common probability distributions, useful for statistics and quantitative biology problems.

Statistics distribution functions: function and distribution.

Function	Distribution
<code>urand(mean, sd, N, digits)</code>	Normal distribution with given mean and standard deviation.
<code>exprand(lambda, N, digits)</code>	Exponential distribution with rate parameter lambda.
<code>poissonrand(lambda, N)</code>	Poisson distribution with rate parameter lambda.
<code>binomrand(p, N, num)</code>	Binomial distribution with probability p and N trials.
<code>bernoullirand(p, num, options)</code>	Bernoulli distribution (single trial, probability p).
<code>discreterand(n, @table)</code>	Discrete distribution from a probability table.

Manual PGRandom: when and how

Most problems do not need a manual PGRandom object. Only create one when you need multiple independent random streams or explicit control over seeding for complex multi-stage randomization.

```
# CORRECT: create an instance, then call srand on the instance.
$local_random = PGRandom->new();
$local_random->srand($problemSeed);

# Use the instance methods for random values.
$value = $local_random->random(1, 10, 1);
$item = $local_random->list_random('A', 'B', 'C');
```

Common PGRandom mistakes

Common PGRandom mistakes: mistake and why it fails.

Mistake	Why it fails
<code>PGRandom->seed(\$problemSeed)</code>	The method is called <code>srand</code> , not <code>seed</code> .
<code>PGRandom->srand(\$problemSeed)</code>	Cannot call <code>srand</code> as a class method. Create an instance with <code>PGRandom->new()</code> first.
<code>SRAND(\$problemSeed)</code>	Resets the global RNG. Usually unnecessary because it is already seeded.

Practical tips

- **Sort hash keys before random selection.** Hash key order is not guaranteed in Perl. Always sort keys first so the same seed yields the same result across runs.
- **Avoid NchooseK and shuffle.** These are deprecated in the PG 2.17 subset. Use built-in random selection or manual array manipulation for subsets.
- **Record the seed when debugging.** Use `$problemSeed` to reproduce failures exactly.

```
# Sort hash keys before selecting randomly.
@sorted_keys = sort keys %hash;
$selected = list_random(@sorted_keys);
```

Guard patterns

Always add guard checks so randomization never produces impossible cases.

Zero denominators

```
$den = random(1, 9, 1);  
if ($den == 0) { $den = 1; }  
$ratio = $num / $den;
```

Repeated values

```
$a = random(1, 10, 1);  
do { $b = random(1, 10, 1); } until ($b != $a);
```

Invalid domains

```
# Ensure the argument to log is positive.  
$conc = random(1, 100, 1);  
if ($conc <= 0) { $conc = 1; }  
$pH = -log($conc / 1000) / log(10);
```

Negative concentrations

```
$stock = random(10, 100, 5);  
$dilution = random(2, 20, 1);  
$final = $stock / $dilution;  
if ($final < 0.1) {  
    $dilution = 2;  
    $final = $stock / $dilution;  
}
```

Biology-specific randomization examples

Randomized dilution series

```
$stock_mM = random(10, 50, 5);  
$dilution_factor = list_random(2, 5, 10, 20);  
$final_uM = ($stock_mM * 1000) / $dilution_factor;
```

Randomized genetics cross

```
# Monohybrid cross with valid Mendelian ratios.  
@dominant_traits = ('Tall', 'Round', 'Purple', 'Axial');  
@recessive_traits = ('Short', 'Wrinkled', 'White', 'Terminal');  
$idx = random(0, scalar(@dominant_traits) - 1, 1);  
$dominant = $dominant_traits[$idx];  
$recessive = $recessive_traits[$idx];  
$total_offspring = random(40, 120, 4);  
$expected_dominant = 3 * $total_offspring / 4;  
$expected_recessive = $total_offspring / 4;
```

```
# qPCR problem with plausible Ct values and efficiency.
$efficiency = random(90, 100, 1) / 100;
$Ct_control = random(18, 25, 1);
$fold_change = list_random(2, 4, 8, 16);
$delta_Ct = -log($fold_change) / log(1 + $efficiency);
$Ct_treated = $Ct_control + $delta_Ct;
```

Complete working example

Randomized genetics problem -- A full problem demonstrating `random()` and `list_random()` with biology content and a guard check to prevent repeated values.

```
## TITLE('Randomized genetics offspring count')
## DESCRIPTION
## Calculate the expected number of offspring showing the
## dominant phenotype from a randomized monohybrid cross.
## ENDDescription
## KEYWORDS('genetics','randomization','numeric','list_random')
## DBsubject('Genetics')
## DBchapter('Mendelian Genetics')
## DBsection('Expected Ratios')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");
@traits = ('Tall', 'Round', 'Purple', 'Axial');
$trait = list_random(@traits);
$total = random(40, 120, 4);
$expected_dom = Real(3 * $total / 4);

BEGIN_PGML
In a monohybrid cross between two heterozygous plants,
[$total] offspring are produced. The dominant trait is
"[$trait]".

How many offspring are expected to show the dominant
phenotype? [_____]{$expected_dom}
```

```
END_PGML
```

```
ENDDOCUMENT ( ) ;
```

Apply it today

- Use the built-in `random()` functions for most problems. They are already seeded.
- Only create a `PGRandom` instance when you need independent random streams.
- Add guard checks so randomization never produces impossible cases.
- Sort hash keys before random selection for deterministic results.

This page titled [6.4: Advanced Randomization Techinques](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.5: Randomized Matching Problems

This page covers modern matching problems using PopUp widgets and two-column layouts. It also explains the one case where `MODES(HTML => ...)` is required: wrapper HTML that must be emitted from inside Perl eval blocks that build dynamic HTML with `join()`.

Use this page when: you need a two-column matching layout with consistent labels and randomized ordering. For basic matching with PopUp dropdowns, see [Matching](#).

When to use matching vs other question types

Matching problems work best for:

- **Definitions:** Match terms with their meanings (e.g., genotype vs phenotype)
- **Associations:** Connect related concepts (e.g., enzyme names with their substrates)
- **Component identification:** Link structures to functions (e.g., organelles to their roles)
- **Classification:** Match examples to categories (e.g., bond types to molecular scenarios)

Not ideal for

- Pure memorization with no conceptual pairing
- Sequence ordering (use drag-and-drop or ordered list instead)
- True/false evaluation of statements (use RadioButtons per statement, see [Randomized MC True and False Statements](#))

Typical problem sizes: 4-8 pairs work well. Fewer than 4 is too easy; more than 8 becomes cognitively overwhelming and increases guessing.

PopUp widgets for matching

Use `PopUp` widgets from `parserPopUp.pl` to create dropdown menus in the left column. Note that `DropDown` is a PG 2.18+ macro and is not available in the PG 2.17 subset used by ADAPT/renderer.

Widget creation pattern

```
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserPopUp.pl",
    "PGcourse.pl",
);

# Define choices (labels A, B, C, D for 4 answers)
my @choices = ('A', 'B', 'C', 'D');

# Define correct answers (mapped to choices)
my @correct_answers = ('B', 'D', 'A', 'C'); # Correct letter for each question

# Create PopUp widgets
my @answer_dropdowns = map { PopUp(\@choices, $correct_answers[$_]) } 0 .. $#correct_answers;
```

How shuffle arrays work

Use the `shuffle()` function to randomize the order of right-column answers while maintaining correct left-right mapping:

```
my @questions = (  
    'Two oxygen atoms share electrons equally',  
    'Sodium transfers an electron to chlorine',  
    'Partial charges on C-O bond',  
    'O-H bond attracted to nearby C=O',  
);  
  
my @answers = (  
    'non-polar covalent bond',  
    'ionic bond',  
    'polar covalent bond',  
    'hydrogen bond',  
);  
  
# Shuffle the right column  
my @shuffle = shuffle(scalar(@answers)); # Returns a permutation [0..3]  
  
# Now answers appear in shuffled order, but @shuffle maps them back to originals
```

Answer evaluation in PGML

In the left column, use `[_]{$answer_dropdowns[$i]}` to insert each PopUp widget. The student selects a letter from the dropdown, and WeBWorK checks it against the correct answer stored in the widget.

Two layout patterns: When MODES is needed

There are two ways to build matching layouts: with wrapper tags directly in PGML (preferred), or with `MODES` when wrapper HTML must be emitted from inside Perl eval blocks.

Pattern 1: Wrapper tags directly in PGML (preferred)

If you can write the wrapper `<div>` tags directly in PGML, do that. This avoids `MODES` entirely and is easier to read and maintain.

```
BEGIN_PGML  
Match each molecular scenario with its bond type.  
  
<div class="two-column">  
<div>  
[@ join("\n\n", map {  
    '[_]{$answer_dropdowns[' . $_ . ']} '  
    . '*' . ($_ + 1) . '.* '  
    . $questions[$_]  
} 0 .. $#questions) @]**  
</div>  
  
<div>  
[@ join("\n\n", map {  
    '*' . chr(65 + $_) . '.* '  
    . $answers[$shuffle[$_]]  
}]
```

```
} 0 .. $#answers) @]**
</div>
</div>
END_PGML
```

In this pattern, the `<div class="two-column">` wrapper is written directly in PGML. The `join()` expressions build the content for each column, and PGML processes everything correctly without needing `MODES`.

Pattern 2: MODES when wrapper must be in eval block (required in specific cases)

Use `MODES(HTML => ...)` only when the wrapper tags must be built inside a Perl eval block, such as when the wrapper and content are created together in a single Perl variable. Without `MODES`, the wrapper HTML gets escaped by PGML and the layout collapses to a single column.

```
# Build left column HTML
my $left_html = join("\n\n", map {
    '[_]{$answer_dropdowns[' . $_ . ']} '
    . '*' . ($_ + 1) . '.* '
    . $questions[$_]
} 0 .. $#questions);

# Build right column HTML
my $right_html = join("\n\n", map {
    '*' . chr(65 + $_) . '.* '
    . $answers[$shuffle[$_]]
} 0 .. $#answers);

# Wrap with MODES because wrapper tags are in eval block
my $columns = MODES(HTML => "<div class='two-column'><div>\n$left_html\n</div><div>\n$right_html\n</div>");

BEGIN_PGML
[$columns]*
END_PGML
```

In this pattern, the wrapper `<div>` tags are built inside the Perl variable `$columns`, so `MODES` is required to prevent PGML from escaping them.

Pattern 3: Inline MODES for wrapper segments (alternative)

You can also emit wrapper HTML segments using inline `MODES` calls between `join()` expressions:

```
BEGIN_PGML
Match each molecular scenario with its bond type.

[@ MODES(HTML => '<div class="two-column"><div>') @]*
[@ join("\n\n", map { ... } 0 .. $#questions) @]**
[@ MODES(HTML => '</div><div>') @]*
[@ join("\n\n", map { ... } 0 .. $#answers) @]**
```

```
[@ MODES(HTML => '</div></div>') @]*  
END_PGML
```

This is a middle-ground approach. Use it when you want the wrapper segments close to the content but cannot write them directly as plain HTML in PGML.

Why MODES is needed (technical explanation)

PGML parses content once and escapes HTML tags for safety. When wrapper `<div>` tags are inside Perl variables or eval blocks, PGML sees them as data to be escaped, not markup. Using `MODES(HTML => ...)` tells PG to emit the HTML directly without escaping, but only in HTML output mode (not TeX/PDF mode).

What happens if you forget MODES: The wrapper `<div>` tags appear as literal text in the rendered problem (e.g., `<div class="two-column">`) or are stripped entirely, causing the layout to collapse to a single column.

CSS styling for two-column layout

Use `HEADER_TEXT` with flexbox CSS to create a responsive two-column layout. This style works across devices and wraps to a single column on narrow screens.

Recommended flexbox style

```
HEADER_TEXT(MODES(HTML => <<'END_STYLE'));  
<style>  
  .two-column {  
    display: flex;  
    flex-wrap: wrap;  
    gap: 2rem;  
    align-items: flex-start;  
    justify-content: space-evenly;  
  }  
  .two-column > div {  
    flex: 1 1 40%;  
    min-width: 250px;  
  }  
</style>  
END_STYLE
```

CSS property explanations

- `display: flex;` - Creates a flexible container for the two columns
- `flex-wrap: wrap;` - Allows columns to wrap on narrow screens (mobile-friendly)
- `gap: 2rem;` - Adds spacing between columns
- `align-items: flex-start;` - Aligns columns to the top (not vertically centered)
- `justify-content: space-evenly;` - Distributes horizontal space evenly
- `flex: 1 1 40%;` - Each column takes ~40% width, can grow or shrink
- `min-width: 250px;` - Prevents columns from becoming too narrow

Alternative alignment options

- `align-items: center;` - Vertically center columns (use if heights differ significantly)
- `justify-content: space-between;` - Push columns to edges with space between
- `gap: 1rem;` - Reduce spacing for compact layouts

Colored labels for fixed label-to-meaning mapping

When answer labels need fixed colors by meaning (not by position), build an HTML mapping in Perl and emit the labels with the `[$var]*` pattern (trailing asterisk required) so PGML does not escape the `<span>` tags.

Building the color mapping

```
# Label -> HTML mapping (fixed colors by meaning)
my %answer_html = (
  'polar covalent bond' => '<span style="color:#00b3b3; font-weight:700;">polar covalent bond</span>',
  'non-polar covalent bond' => '<span style="color:#b3b300; font-weight:700;">non-polar covalent bond</span>',
  'ionic bond' => '<span style="color:#009900; font-weight:700;">ionic bond</span>',
  'hydrogen bond' => '<span style="color:#e60000; font-weight:700;">hydrogen bond</span>';
);

# Build colored HTML labels array
my @answers_html = map { $answer_html{$_} } @answers;
```

Emitting colored labels in PGML

In the right-column `join()` expression, use `[$answers_html[$shuffle[$_]]]*` with the trailing asterisk to pass through the HTML without escaping:

```
[@ join(
  "\n\n",
  map {
    '*' . chr(65 + $_) . '.* '
      . '[$answers_html[$shuffle[' . $_ . ']]]*'
  } 0 .. $#answers
) @]**
```

Why the trailing asterisk is required

Without the trailing `*`, PGML escapes HTML tags for safety. Using `[$var]*` tells PGML to pass through the variable's content as raw HTML. This keeps colors tied to label meaning even when the order is randomized. For more on this pattern, see [Text Coloring and Emphasis](#).

Biology examples

Example 1: Chemical bond types

Scenario: Match molecular interactions with the correct bond type. Use colored labels to distinguish bond types visually.

```
my @questions = (
  'Two oxygen atoms share electrons equally',
  'Sodium transfers an electron to chlorine',
  'Partial charges on C-O bond',
  'O-H bond attracted to nearby C=O',
);

my @answers = (
  'non-polar covalent bond',
```

```
'ionic bond',
'polar covalent bond',
'hydrogen bond',
);

my @shuffle = shuffle(scalar(@answers));

my %answer_html = (
  'non-polar covalent bond' => '<span style="color:#b3b300; font-weight:700;">non-polar covalent bond</span>',
  'ionic bond' => '<span style="color:#009900; font-weight:700;">ionic bond</span>',
  'polar covalent bond' => '<span style="color:#00b3b3; font-weight:700;">polar covalent bond</span>',
  'hydrogen bond' => '<span style="color:#e60000; font-weight:700;">hydrogen bond</span>',
);
my @answers_html = map { $answer_html{$_} } @answers;
```

Note: Fixed colors ensure that "hydrogen bond" is always red, regardless of randomization, helping students build consistent mental associations.

Example 2: Pathway components matched to functions

Scenario: Match glycolytic enzymes with their specific reactions in the pathway.

```
my @questions = (
  'Converts glucose-6-phosphate to fructose-6-phosphate',
  'Phosphorylates fructose-6-phosphate (rate-limiting step)',
  'Splits fructose-1,6-bisphosphate into two 3-carbon sugars',
  'Converts phosphoenolpyruvate to pyruvate (ATP-generating step)',
);

my @answers = (
  'phosphoglucose isomerase',
  'phosphofructokinase-1',
  'aldolase',
  'pyruvate kinase',
);

my @choices = ('A', 'B', 'C', 'D');
my @shuffle = shuffle(scalar(@answers));

my @correct_answers = map { $choices[$shuffle[$_]] } 0 .. $#questions;
my @answer_dropdowns = map { PopUp(\@choices, $correct_answers[$_]) } 0 .. $#questions;
```

Note: This example demonstrates function-to-enzyme matching without colors, focusing on conceptual pairing for pathway memorization.

Example 3: Genetics terminology matched to definitions

Scenario: Match genetics terms with their formal definitions.

```
my @questions = (
    'Observable physical characteristic',
    'Genetic makeup at a locus',
    'Alternative form of a gene',
    'Physical location of a gene on a chromosome',
);

my @answers = (
    'phenotype',
    'genotype',
    'allele',
    'locus',
);

my @choices = ('A', 'B', 'C', 'D');
my @shuffle = shuffle(scalar(@answers));

my @correct_answers = map { $choices[$shuffle[$_]] } 0 .. $#questions;
my @answer_dropdowns = map { PopUp(\@choices, $correct_answers[$_]) } 0 .. $#questions;
```

Note: This is a pure definition-matching problem, ideal for introducing formal terminology early in a genetics unit.

Common failures and fixes

Failure 1: Forgot trailing asterisk on colored labels

Symptom: Raw `<span>` tags appear in the rendered problem instead of colored text.

Cause: Used `[$var]` instead of `[$var]*`, so PGML escaped the HTML for safety.

Fix: Always use `[$answers_html[$shuffle[$_]]]*` with the trailing asterisk to pass through HTML.

Prevention: Check all variable interpolations that contain HTML. The rule is: `[$var]*` for HTML, `[$var]` for plain text.

Failure 2: Used MODES for everything

Symptom: Code works but is unnecessarily complex and harder to maintain.

Cause: Wrapped every HTML snippet in `MODES`, including cases where wrapper tags could be written directly in PGML.

Fix: Use Pattern 1 (wrapper tags directly in PGML) whenever possible. Only use `MODES` when the wrapper HTML is inside Perl eval blocks.

Prevention: Ask: "Can I write these `<div>` tags directly in PGML?" If yes, do that.

Failure 3: Shuffle array index mismatch

Symptom: Wrong answers are marked correct. Dropdown selections don't map to the displayed right-column labels.

Cause: Used `$shuffle[$_]` incorrectly in left column or right column, breaking the mapping.

Fix: Verify that left column uses `$answer_dropdowns[$_]` (no shuffle) and right column uses `$answers[$shuffle[$_]]` (with shuffle).

Prevention: Test with `problemSeed` values that produce different shuffle orders. Check that correct answers remain correct.

Failure 4: PopUp choices don't match answer labels

Symptom: Dropdown shows letters that don't appear in the right column, or vice versa.

Cause: Used different `@choices` arrays for PopUp creation and right-column labels.

Fix: Use the same `@choices` array (e.g., `('A', 'B', 'C', 'D')`) for both PopUp widget creation and right-column label generation.

Prevention: Define `@choices` once at the top of setup and reference it everywhere.

Failure 5: Layout collapses to single column

Symptom: Both columns stack vertically instead of appearing side-by-side.

Cause: Wrapper `<div>` tags were built in Perl without `MODES`, so PGML escaped them.

Fix: Either move wrapper tags directly into PGML (Pattern 1) or wrap them with `MODES(HTML => ...)` (Pattern 2).

Prevention: If wrapper HTML is inside a Perl variable, use `MODES`. If it's directly in PGML, no `MODES` needed.

Creating matching banks with YAML

Instead of writing each matching problem by hand, you can define a bank of matching pairs in a YAML file and use a generator script to produce PGML files automatically. This workflow separates content authoring from code, making it easier for instructors to contribute questions without editing Perl.

File naming convention

Use filenames that are readable to instructors and map naturally to course chapters. Follow the pattern: topic prefix + plain-English descriptor.

- `enzymes_basics.yml` -- enzyme classes matched to reaction types
- `senses_receptor_types.yml` -- receptor types matched to sensory modalities
- `metabolism_pathway_components.yml` -- pathway enzymes matched to functions

Avoid deep jargon in filenames (keep jargon inside the YAML content). Future-you should know what the bank covers without opening the file.

YAML file structure

Every YAML bank requires four top-level keys that describe what students are matching:

YAML file structure: key, purpose, and example.

Key	Purpose	Example
<code>key description</code>	Singular label for the left column	<code>enzyme class</code>
<code>keys description</code>	Plural label for the left column	<code>enzyme classes</code>
<code>value description</code>	Singular label for the right column	<code>reaction type</code>
<code>values description</code>	Plural label for the right column	<code>reaction types</code>

The fifth required key is `matching pairs`, which holds the actual content (see next section).

Matching pairs format

Under `matching pairs`, each key maps to a list of one or more values. Each key is a left-column concept; each value is a right-column definition or function.

```
matching pairs:
  Kinase:
```

- Adds a phosphate group to a substrate (phosphorylation)
 - Transfers a phosphate group to a molecule
- Phosphatase:
- Removes a phosphate group from a substrate (dephosphorylation)
 - Catalyzes phosphate removal from a molecule

Each key must map to a YAML list, even if there is only one value. For each generated question, the generator picks one value per key at random.

Same concept, different phrasing

The most important quality rule: put synonyms and equivalent statements under one key as multiple values. Do not split a single concept across multiple keys (that turns synonyms into trick questions).

This makes banks feel fresh without changing what you are assessing. Students see one wording per key in any single question, but across different generated questions the bank naturally rotates through alternate wordings.

```
matching pairs:
  Protease:
    - Breaks peptide bonds in proteins
    - Hydrolyzes peptide bonds
  Ligase:
    - Joins two molecules by forming a new bond (often using ATP)
    - Catalyzes bond formation to link molecules
```

Excluding confusable pairs

If two keys are too similar (or their values are too easy to mix up), use `exclude pairs` so they never appear together in the same question:

```
exclude pairs:
  - [Coenzyme, Cofactor]
  - [competitive inhibitor, non-competitive inhibitor]
```

Use `exclude pairs` sparingly, only for classic confusions where students can reasonably know both terms yet still mix them up under time pressure. Do not exclude pairs just because two terms are related -- that can remove the learning objective.

Replacement rules for formatting and color

The `replacement_rules` key defines simple search-and-replace substitutions applied after question text is constructed. This is commonly used for highlighting phrases with color or ensuring consistent typography.

```
replacement_rules:
  releases&nbsp;energy: "<span style='color: #e60000;'>releases&nbsp;energy</span>"
```

Colors are assigned deterministically: the same prompt set gets the same colors on every run. If any prompt already contains HTML color markup, automatic coloring is disabled for the entire question so author-chosen colors are preserved.

Optional metadata keys

The generators ignore unknown top-level keys, so you can add author-facing metadata for long-term maintenance:

- `topic` -- short label (e.g., `phototransduction`)
- `learning_objective` -- one sentence (e.g., `Trace the cGMP pathway in rods`)

- `source` -- textbook or chapter reference
- `items to match per question` -- how many pairs appear per question (default 5)

Generating PGML files from YAML

The `yaml_match_to_pgml.py` script reads a YAML bank and produces matching-problem PGML files. The generated files follow the same PopUp widget and `MODES` wrapper patterns described earlier in this chapter.

What the generator does

- Reads the YAML matching pairs and descriptions
- Selects one value per key at random (rotating through phrasings across runs)
- Applies replacement rules for color and formatting
- Unescapes HTML entities (named and numeric) into Unicode for rendering without MathJax
- Converts `<sub>` / `<sup>` tags into Unicode subscripts/superscripts
- Outputs PGML files using the two-column flexbox layout with PopUp widgets

The default output uses inline HTML spans for replacement-rule colors. Use `--no-color` to disable styling.

Quality checks for matching banks

Before shipping a new matching set, run through these checks:

Quality checks for matching banks: check and what to look for.

Check	What to look for
Single learning objective per key	Each key should test one concept, not a bundle of related facts
Parallel grammar	All values should use the same grammatical form (all noun phrases, or all full sentences)
Similar value length	If one answer is much longer than others, students use length as a hint
No keyword cues	If a prompt contains a distinctive word, do not repeat it in the matching value
Distinct values across keys	If multiple keys could match the same value text, students are guessing
Accessibility	Do not rely on color alone to convey meaning; use color as an accent

Difficulty control

Adjust the difficulty of matching questions by changing the number of items and the phrasing style:

- **Easier:** Use direct definitions and unique vocabulary. Set `items to match per question: 3` or `4` for fewer items per question.
- **Harder:** Use mechanism or consequence phrasing instead of definitions. Include more similar terms in the bank, then use `exclude pairs` for the worst confusions.

Copy/paste YAML example: enzyme classes

This is a complete, working YAML file for a matching bank. Copy it as a starting point for new banks.

```
#YAML file for Matching Questions
topic: enzymes_basics
learning_objective: Match enzyme classes to what they do.
```

items to match per question: 5

key description: enzyme class

keys description: enzyme classes

value description: reaction type

values description: reaction types

matching pairs:

Kinase:

- Adds a phosphate group to a substrate (phosphorylation)
- Transfers a phosphate group to a molecule

Phosphatase:

- Removes a phosphate group from a substrate (dephosphorylation)
- Catalyzes phosphate removal from a molecule

Protease:

- Breaks peptide bonds in proteins
- Hydrolyzes peptide bonds

Ligase:

- Joins two molecules by forming a new bond (often using ATP)
- Catalyzes bond formation to link molecules

Isomerase:

- Rearranges atoms within a molecule to form an isomer
- Converts a molecule into its isomer

Each enzyme class key has two phrasing variants. The generator picks one wording per question, so students see different phrasings across attempts while always being assessed on the same concept.

Complete working example

Organelle function matching -- A full problem using PopUp dropdowns to match organelles with their primary functions.

```
## TITLE('Match organelles to functions')
## DESCRIPTION
## Match each organelle with its primary function using
## PopUp dropdown widgets.
## ENDDescription
## KEYWORDS('organelles', 'matching', 'PopUp', 'cell biology')
## DBsubject('Cell Biology')
## DBchapter('Organelles')
## DBsection('Function')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
```

```
"PGML.pl",
"parserPopUp.pl",
"PGcourse.pl",
);

@functions = ('ATP production', 'Protein sorting',
             'Intracellular digestion', 'Lipid synthesis');

$popup1 = PopUp(['?', @functions], 'ATP production');
$popup2 = PopUp(['?', @functions], 'Protein sorting');
$popup3 = PopUp(['?', @functions], 'Intracellular digestion');
$popup4 = PopUp(['?', @functions], 'Lipid synthesis');

BEGIN_PGML
Match each organelle with its primary function.

1. Mitochondrion [_]{$popup1}

2. Golgi apparatus [_]{$popup2}

3. Lysosome [_]{$popup3}

4. Smooth ER [_]{$popup4}
END_PGML

ENDDOCUMENT();
```

Apply it today

- Write wrapper `<div>` tags directly in PGML whenever possible (Pattern 1).
- Use `MODES(HTML => ...)` only for wrapper HTML built inside eval blocks (Pattern 2).
- Always use `[$var]*` (trailing asterisk) for HTML variables to prevent escaping.
- Define `@choices` once and use it for both PopUp widgets and right-column labels.
- Test with multiple `problemSeed` values to verify shuffle mapping works correctly.
- Use 4-8 pairs for optimal cognitive load and assessment effectiveness.
- For YAML-based banks, define 8-20 keys with 2-4 phrasing variants each.
- Run quality checks for parallel grammar, similar value length, and no keyword cues.
- For more on colored emphasis, see [Text Coloring and Emphasis](#). For statement-based questions, see [Randomized MC True and False Statements](#).

This page titled [6.5: Randomized Matching Problems](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.6: Randomized MC True and False Statements

This page shows how to build statement-by-statement evaluation prompts using RadioButtons or PopUp widgets for each statement. It also covers how to create YAML-based statement banks and generate multiple-choice "Which statement is TRUE/FALSE?" questions from them.

Use this page when: you want students to mark each statement as true/false or correct/incorrect rather than choose one best answer, or when you want to generate multiple-choice questions from curated pools of true and false statements. For basic single-answer multiple choice, see [Multiple Choice](#). For checkbox-style multiple answer, see [Multiple Answer](#).

When to use MC statements vs matching sets

Statement-based multiple choice is best when you want students to **evaluate ideas** (truth or falsehood), not just map terms to definitions.

Good fits for MC statements:

- "Which statement is TRUE/FALSE?" conceptual checks
- Misconception diagnostics (wrong molecule, wrong direction, wrong location, sign flips)
- "Why" and "what changes" logic that is hard to make one-to-one

If your goal is to map terms or components to their matching definitions or functions (especially pathways and part-to-function), consider a matching set instead (see [Randomized Matching Problems](#)).

Copy and edit (RadioButtons per statement)

```
$tf = [ 'True', 'False' ];
$rb1 = RadioButtons($tf, 'True');
$rb2 = RadioButtons($tf, 'False');

BEGIN_PGML
1. The enzyme activity increases with temperature. [_]{$rb1}

2. The inhibitor decreases Vmax in competitive inhibition. [_]{$rb2}
END_PGML
```

Common failure and fix

```
Broken: parserCheckboxList.pl used for statement checks
What you will see: widget missing or render failure
Fix: use RadioButtons or PopUp per statement
```

Styling and emphasis

- Use HTML spans to emphasize key words like NOT or ONLY.
- Keep each statement short so students can scan quickly.

Creating statement banks with YAML

Instead of hand-coding each PGML problem, you can define pools of true and false statements in a YAML file and use a generator script to produce multiple-choice questions automatically. Each generated question presents one correct answer among several distractors drawn from the bank.

YAML file structure

Each YAML file requires three top-level keys:

YAML file structure: key, required, and description.

Key	Required	Description
<code>topic</code>	Yes	Short phrase used in the question stem (e.g., <code>DNA structure</code>)
<code>true_statements</code>	Yes	Mapping of statement IDs to correct statement text
<code>false_statements</code>	Yes	Mapping of statement IDs to incorrect statement text

Common optional keys

Common optional keys: key and description.

Key	Description
<code>connection_words</code>	YAML list of stem connectors (defaults to built-in words like <code>concerning</code> , <code>about</code> , <code>regarding</code>)
<code>replacement_rules</code>	Mapping of literal substring replacements for consistent formatting or color
<code>override_question_true</code>	Replace the TRUE stem with custom text, or set to <code>~</code> to disable TRUE questions
<code>override_question_false</code>	Replace the FALSE stem with custom text, or set to <code>~</code> to disable FALSE questions

In YAML, `~` means null (like Python `None`). For `connection_words` , null means "use the built-in defaults." For override keys, null means "disable that question form entirely."

Statement IDs and conflict groups

Statement IDs follow the pattern `truth<N>a` , `truth<N>b` , `false<N>a` , `false<N>b` , and so on. Statements that share the same number represent the **same concept with different phrasing**. The generator automatically prevents variants of the same concept from appearing together in one question.

```
true_statements:
  truth3a: Enzymes lower activation energy.
  truth3b: Enzymes speed reactions by lowering activation energy.
```

Because `truth3a` and `truth3b` share the number 3, they will never both appear as choices in the same generated question. This lets you author multiple equivalent phrasings for the same idea without creating confusing near-duplicate answer choices.

Replacement rules for formatting consistency

The `replacement_rules` key is a simple search-and-replace table applied to all question text and choices. It is commonly used for:

- Highlighting key terms with HTML color spans
- Consistent symbols and typography (e.g., `ΔG` , ` ` ;)

```
replacement_rules:
  cAMP: "<strong>cAMP</strong>"
```

```
cGMP: "<strong>cGMP</strong>"
```

Generating PGML files from YAML

The `yaml_mc_statements_to_pgml.py` script reads the same YAML schema described above and generates WeBWorK PGML problem files. This workflow uses the same statement bank format, so you can maintain one YAML file and produce both Blackboard export files and PGML problems from it.

```
source source_me.sh
python3 yaml_mc_statements_to_pgml.py -y your_set.yml -x 20
```

The `-x` flag controls how many question variants to generate. By default, the generator creates both "Which statement is TRUE?" and "Which statement is FALSE?" forms. You can disable either form with `--no-true` or `--no-false`.

Second messenger signaling example (complete YAML)

```
---
topic: second messenger signaling
learning_objective: Distinguish cAMP vs cGMP roles and enzymes that regulate them.

replacement_rules:
  cAMP: "<strong>cAMP</strong>"
  cGMP: "<strong>cGMP</strong>"

true_statements:
  truth1a: cAMP is a second messenger made from ATP
  truth1b: cyclic AMP (cAMP) is produced from ATP by adenylyl cyclase
  truth2: phosphodiesterase breaks down cyclic nucleotides to terminate a signal
  truth3: Protein kinase A (PKA) is activated by cAMP

false_statements:
  false1a: cAMP is a second messenger made from glucose
  false1b: cyclic AMP (cAMP) is produced from ADP by phosphodiesterase
  false2: phosphodiesterase makes cAMP from ATP
  false3: Protein kinase A (PKA) is activated by cGMP
```

Note the conflict groups: `truth1a` and `truth1b` both describe how cAMP is made, so they share number 1 and will not appear together. The same applies to `false1a` and `false1b`.

What makes a good false statement

False statements should be "plausible but wrong" in a way that diagnoses a specific misconception. Good patterns include:

- **Wrong substrate or reactant:** ATP vs GTP, NADH vs NADPH
- **Wrong directionality:** influx vs efflux, synthesis vs degradation
- **Wrong cellular location:** nucleus vs cytosol vs mitochondrion
- **Swapped cofactors or ions:** Mg²⁺ vs Ca²⁺
- **Wrong sign or relationship:** reversed thermodynamic logic using delta-G or equilibrium

Practical trick: write the true statement first, then create the false version by changing exactly one detail.

Avoiding accidental cues

Three quick checks that prevent students from using test-taking strategies instead of content knowledge:

- **Similar length:** Do not let one statement be noticeably longer or shorter than the others.
- **Parallel grammar:** Use the same sentence structure across all choices (all noun phrases, or all full sentences).
- **No unique keywords:** Avoid terms that appear in only one choice, as they can give away the answer.

Additional statement hygiene: one main idea per statement (avoid "and also..." bundles), use concrete biology verbs (binds, phosphorylates, opens, inhibits), avoid double negatives, and define abbreviations on first use.

Difficulty control

You can adjust question difficulty by varying how the false statements are constructed:

Difficulty control: difficulty and strategy.

Difficulty	Strategy
Easier	Use direct definitions and unique vocabulary. Keep false statements clearly wrong with one obvious error.
Harder	Make false statements "plausible but wrong" using common misconceptions. Use more similar terms across choices, relying on conflict groups to avoid near-duplicates in the same question.

Quality checks for statement banks

Before using a bank in an assessment, run through this checklist:

- **Misfiled statements:** Confirm all `truth*` entries are under `true_statements` and all `false*` entries are under `false_statements`.
- **ID collisions:** Do not reuse the same ID for two unrelated meanings (e.g., two different `truth5a` ideas).
- **Duplicate text:** The generator fails if the exact same sentence appears twice, even across true and false sections.
- **Replacement rule overlap:** Use specific, non-overlapping phrases to avoid nested spans or partial matches.
- **Grammar scan:** Fix typos and subject/verb agreement. Correct grammar improves student trust even when the biology is right.

Biology examples to model

- Experimental design statements (controls, variables, replication).
- Pathway regulation claims (activation vs repression).

Complete working example

Enzyme true/false statements -- A full problem with per-statement RadioButtons evaluating claims about enzyme behavior.

```
## TITLE('Enzyme behavior true or false')
## DESCRIPTION
## Evaluate true/false statements about enzyme regulation
## and kinetics using per-statement RadioButtons.
## ENDDescription
## KEYWORDS('enzymes','true false','RadioButtons','kinetics')
## DBsubject('Biochemistry')
## DBchapter('Enzymes')
## DBsection('Regulation')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')
```

```
DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$tf = [ 'True', 'False' ];
$rb1 = RadioButtons($tf, 'True');
$rb2 = RadioButtons($tf, 'False');
$rb3 = RadioButtons($tf, 'True');
$rb4 = RadioButtons($tf, 'False');

BEGIN_PGML
For each statement about enzymes, select True or False.

1. Enzymes increase the rate of a reaction without being consumed. [_]{$rb1}

2. A competitive inhibitor decreases the Vmax of an enzyme. [_]{$rb2}

3. Allosteric regulation involves binding at a site other than the active site. [_]{$rb3}

4. Increasing temperature always increases enzyme activity. [_]{$rb4}
END_PGML

ENDDOCUMENT();
```

Apply it today

- Use one RadioButtons object per statement and keep answer keys aligned.
- Avoid checkbox macros not available in the PG 2.17 subset.
- Create YAML statement banks with 8 to 20 items per truth category for good question variety.
- Use conflict group IDs (e.g., `truth4a` , `truth4b`) for equivalent phrasings of the same concept.
- Write the true statement first, then create the false version by changing one detail.
- Run the quality checklist before deploying a bank to an assessment.

This page titled [6.6: Randomized MC True and False Statements](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.7: Rendering Chemical Structures with RDKit

This page explains how to render publication-quality chemical structures in WeBWorK problems using RDKit.js and SMILES strings. Students see interactive, consistent molecular drawings without any image files to manage.

Use this page when: you want to display chemical structures such as amino acids, nucleotides, or peptides in a PGML problem. RDKit.js is loaded via `HEADER_TEXT` (see [Common PG Macros](#) for macro loading conventions). For common RDKit rendering errors, see [Common Mistakes and How to Fix Them](#).

Why RDKit.js instead of static images

- High-quality, consistent rendering across all molecules and devices.
- No image file management or hosting required.
- SMILES strings are compact and easy to maintain in PG code.
- Customizable rendering options (explicit hydrogens, stereochemistry labels, legends).
- Students cannot reverse-image-search for answers.

RDKit.js loads from a public CDN and runs entirely in the browser. It requires an internet connection and a modern browser with JavaScript enabled.

Quick start pattern

Every RDKit problem follows three steps: define a SMILES string, create an HTML canvas element, and load RDKit.js through `HEADER_TEXT`. Below is a minimal working example that displays L-alanine.

Step 1: Define SMILES and create a canvas

```
# -----  
# 2) Setup  
# -----  
  
# SMILES for L-alanine (zwitterion at physiological pH)  
$smiles_alanine = 'C[C@@H](C(=O)[O-])[NH3+]';  
  
# Build the canvas element with a unique id  
$canvas_alanine = '<canvas id="canvas_alanine" width="320" height="240" '  
    . ' data-smiles="' . $smiles_alanine . '"></canvas>';
```

Step 2: Load RDKit.js in `HEADER_TEXT`

Place this block after `loadMacros` but before `BEGIN_PGML`. The JavaScript finds every canvas that has a `data-smiles` attribute and draws the corresponding molecule.

```
HEADER_TEXT(<<'END_HEADER');  
<script src="https://unpkg.com/@rdkit/rdkit/dist/RDKit_minimal.js"></script>  
<script>  
let RDKitReady = null;  
function getRDKit() {  
    if (!RDKitReady) {  
        RDKitReady = initRDKitModule();  
    }  
    return RDKitReady;  
}
```

```
function initMoleculeCanvases() {
  getRDKit().then(function(RDKit) {
    const canvases = document.querySelectorAll('canvas[data-smiles]');
    canvases.forEach((canvas) => {
      const smiles = canvas.dataset.smiles;
      if (!smiles) { return; }
      const mol = RDKit.get_mol(smiles);
      const mdetails = {
        "legend": "",
        "explicitMethyl": true
      };
      if (canvas && mol) {
        mol.draw_to_canvas_with_highlights(
          canvas, JSON.stringify(mdetails)
        );
        mol.delete();
      }
    });
  }).catch(error => {
    console.error('Error initializing RDKit:', error);
  });
}
if (document.readyState === 'loading') {
  document.addEventListener('DOMContentLoaded', initMoleculeCanvases);
} else {
  initMoleculeCanvases();
}
</script>
END_HEADER
```

Step 3: Display the canvas in PGML

```
BEGIN_PGML
**Identify this amino acid:**

[$canvas_alanine]*

[_]{$answer}
END_PGML
```

The `*` after the closing bracket tells PGML not to escape the HTML, so the browser sees a real `<canvas>` element instead of literal tags.

SMILES basics

SMILES (Simplified Molecular Input Line Entry System) is a compact text notation for chemical structures. If you have never used SMILES before, the rules below are all you need for biochemistry problems.

Feature	Syntax	Example
Standard atoms	Uppercase letters: C, N, O, S, P	<chem>CCO</chem> (ethanol)
Single bond	Implicit (omitted between atoms)	<chem>CC</chem> (ethane)
Double bond	<code>=</code>	<chem>C=O</chem> (formaldehyde core)
Triple bond	<code>#</code>	<chem>C#N</chem> (cyanide)
Branches	Parentheses <code>(...)</code> for side chains	<chem>CC(=O)O</chem> (acetic acid)
Rings	Matching digits to close a ring	<chem>C1CCCCC1</chem> (cyclohexane)
Aromatic atoms	Lowercase letters	<chem>c1ccccc1</chem> (benzene)

Stereochemistry

Many biological molecules are chiral. SMILES encodes chirality at tetrahedral centers with [C@@H] and [C@H], and double-bond geometry with `/` and `\`. For amino acids, the L-form typically uses [C@@H].

```
# L-Alanine breakdown:
#  C          methyl group (CH3)
#  [C@@H]     chiral center (L-configuration)
#  (C(=O)[O-]) carboxylate branch
#  [NH3+]     protonated amino group

$ala = 'C[C@@H](C(=O)[O-])[NH3+]';
```

Charged species (critical for biochemistry)

Most biological molecules carry formal charges at physiological pH. SMILES uses square brackets [...] for atoms with formal charges, explicit hydrogens, or non-default valence. Getting charges right is the single most common source of errors in biochemistry SMILES.

Common ionizable groups

Common ionizable groups: group, neutral form, charged form, and more.

Group	Neutral form	Charged form	SMILES (neutral)	SMILES (charged)
Carboxylic acid	COOH	COO-	<chem>C(=O)O</chem>	<chem>C(=O)[O-]</chem>
Amino group	NH ₂	NH ₃ ⁺	<chem>N</chem>	<chem>[NH3+]</chem>
Imidazole (His)	NH + N	NH + NH ⁺	<chem>C1=CN=C([NH]1)</chem>	<chem>C1=C([NH]C=[NH+]1)</chem>
Guanidinium (Arg)	C(=NH)(NH ₂)NH ₂	C(=NH ₂ ⁺)(NH ₂)NH ₂	<chem>C(=N)(N)N</chem>	<chem>C(=[NH2+])(N)N</chem>
Thiol (Cys)	SH	S-	<chem>S</chem>	<chem>[S-]</chem>
Phenol (Tyr)	OH	O-	<chem>O</chem>	<chem>[O-]</chem>

Critical note: imidazole protonation

The imidazole ring on histidine is the most common source of SMILES errors. The protonated form has **one hydrogen on each nitrogen** (NH + NH⁺), not two hydrogens on one nitrogen. Using [NH2+] is incorrect for imidazole because that implies two

hydrogens on a single nitrogen.

```
# CORRECT: protonated histidine (pH below 6)
# Imidazole ring has NH and NH+ (one H on each nitrogen)
$this_protonated = 'C1=C([NH]C=[NH+]1)C[C@@H](C(=O)[O-])[NH3+]';

# CORRECT: neutral histidine (pH above 6)
# Imidazole ring has NH and N (hydrogen on only one nitrogen)
$this_neutral = 'C1=CN=C([NH]1)C[C@@H](C(=O)[O-])[NH3+]';

# WRONG: do not use NH2+ for protonated imidazole
# $this_wrong = 'C1=C(N=C[NH2+]1)...';
```

Validating SMILES strings

Always validate every SMILES string before using it in a problem. An invalid SMILES will silently produce an empty canvas. Three methods are available, listed from most to least reliable.

Method 1: Python RDKit (recommended)

```
from rdkit import Chem

smiles = "C[C@@H](C(=O)[O-])[NH3+]"
mol = Chem.MolFromSmiles(smiles)

if mol is None:
    print("ERROR: Invalid SMILES!")
else:
    print(f"Valid SMILES")
    print(f"Formula: {Chem.rdMolDescriptors.CalcMolFormula(mol)}")
    print(f"Charge: {Chem.GetFormalCharge(mol)}")
```

Method 2: Online SMILES viewers

Paste a SMILES string into the PubChem Sketcher (<https://pubchem.ncbi.nlm.nih.gov/edit3/index.html>) or any other online SMILES viewer to see the rendered structure immediately.

Method 3: Test render in WeBWork

Use the local renderer to preview the problem and confirm that the canvas draws the expected molecule.

Canvas sizing guidelines

The canvas dimensions control how large the structure appears. Choose a size based on molecule complexity.

Canvas sizing guidelines: use case, width x height, aspect ratio, and more.

Use case	Width x Height	Aspect ratio	Examples
Standard molecule	320 x 240	4:3	Single amino acids, monosaccharides
Large molecule	480 x 320	3:2	Dipeptides, nucleotides, fatty acids
Very large molecule	640 x 480	4:3	Tripeptides, complex lipids
Compact (grid display)	240 x 180	4:3	Side-by-side comparisons

Rendering options (mdetails)

The `mdetails` object in the JavaScript controls how RDKit draws each molecule. Adjust these settings inside the `HEADER_TEXT` block.

Production settings (for students)

```
const mdetails = {
  "legend": "",
  "explicitMethyl": true
};
```

Debug settings (for development)

```
const mdetails = {
  "legend": "Debug view",
  "explicitMethyl": true,
  "addAtomIndices": true,
  "addBondIndices": true,
  "addStereoAnnotation": true
};
```

Debug settings (for development): option, effect, and default.

Option	Effect	Default
<code>legend</code>	Text label drawn below the molecule.	<code>""</code>
<code>explicitMethyl</code>	Shows CH3 groups as text rather than a bare vertex.	<code>true</code>
<code>addAtomIndices</code>	Numbers each atom (useful for debugging).	<code>false</code>
<code>addBondIndices</code>	Numbers each bond (useful for debugging).	<code>false</code>
<code>addStereoAnnotation</code>	Draws R/S and E/Z labels on stereocenters.	<code>false</code>

Multiple molecules in one problem

Pattern 1: Sequential display

Create a separate canvas for each molecule. Each canvas must have a unique `id`.

```
$smiles_ala = 'C[C@@H](C(=O)[O-])[NH3+]';
$smiles_gly = 'C(C(=O)[O-])[NH3+]';
```

```
$canvas_ala = '<canvas id="canvas_ala" width="320" height="240"'
. ' data-smiles="' . $smiles_ala . '"></canvas>';
$canvas_gly = '<canvas id="canvas_gly" width="320" height="240"'
. ' data-smiles="' . $smiles_gly . '"></canvas>';

BEGIN_PGML
**Molecule A:**
[$canvas_ala]*

**Molecule B:**
[$canvas_gly]*

Which molecule is glycine? [_]{$answer}
END_PGML
```

Pattern 2: Side-by-side with HTML wrapper

Wrap multiple canvases in a flex container to display them in a row.

```
$side_by_side = '<div style="display: flex; gap: 1em;">'
. '<div>' . $canvas_ala . '<p>Molecule A</p></div>'
. '<div>' . $canvas_gly . '<p>Molecule B</p></div>'
. '</div>';

BEGIN_PGML
Compare these molecules:

[$side_by_side]*

Which is glycine? [_]{$answer}
END_PGML
```

Randomization with RDKit

Use a Perl hash to map molecule names to SMILES strings, then randomly select one. Always sort the hash keys before random selection so the same problem seed produces the same molecule.

```
%amino_acids = (
    'alanine' => 'C[C@@H](C(=O)[O-])[NH3+]',
    'glycine' => 'C(C(=O)[O-])[NH3+]',
    'serine'  => 'C([C@@H](C(=O)[O-])[NH3+])O',
);

# Sort keys before random selection for seed stability
@aa_names = sort keys %amino_acids;
$selected_name = list_random(@aa_names);
$selected_smiles = $amino_acids{$selected_name};
```

```
$canvas = '<canvas id="canvas_aa" width="320" height="240" '
. ' data-smiles="' . $selected_smiles . '"></canvas>';

BEGIN_PGML
Identify this amino acid:

[$canvas]*

[_]{$selected_name}
END_PGML
```

Why sort the keys? Perl hashes have no guaranteed ordering. Without sorting, the same seed could map to different molecules on different runs or different servers, making grades unreproducible.

Common pitfalls

1. Incorrect formal charges

```
# WRONG: missing brackets around charged atom
$smiles = 'CC(C(=O)O)NH3+';

# RIGHT: square brackets enclose charged atoms
$smiles = 'CC(C(=O)O)[NH3+]';
```

2. Imidazole protonation errors

```
# WRONG: NH2+ implies two H on one nitrogen
$this = 'C1=C(N=C[NH2+]1)...';

# RIGHT: NH and NH+ (one H on each nitrogen)
$this = 'C1=C([NH]C=[NH+]1)...';
```

3. Quote escaping in canvas attributes

```
# WRONG: nested double quotes break the HTML attribute
$canvas = "<canvas data-smiles=\""$smiles\""></canvas>";

# RIGHT: use single-quoted string with dot concatenation
$canvas = '<canvas data-smiles="' . $smiles . '"></canvas>';
```

4. Canvas ID collisions

```
# WRONG: duplicate IDs cause only one molecule to render
$canvas1 = '<canvas id="canvas" ...></canvas>';
$canvas2 = '<canvas id="canvas" ...></canvas>';

# RIGHT: every canvas gets a unique ID
```

```
$canvas1 = '<canvas id="canvas_mol1" ...></canvas>';  
$canvas2 = '<canvas id="canvas_mol2" ...></canvas>';
```

5. Missing asterisk in PGML

```
# WRONG: PGML escapes the HTML tags  
[$canvas]  
  
# RIGHT: the * tells PGML to output raw HTML  
[$canvas]*
```

Chemistry-specific guidelines

Amino acids

At physiological pH (around 7.4), amino acids exist as zwitterions with a protonated amino group ($\text{[NH}_3^+]$) and a deprotonated carboxylate ($\text{C(=O)[O}^-]$). Side chains may carry additional charges depending on their pKa.

```
# L-Alanine (simple zwitterion)  
$ala = 'C[C@@H](C(=O)[O-])[NH3+]';  
  
# L-Lysine (charged side chain: extra NH3+)  
$lys = 'C(CC[NH3+])[C@@H](C(=O)[O-])[NH3+]';  
  
# L-Aspartate (charged side chain: extra COO-)  
$asp = 'C([C@@H](C(=O)[O-])[NH3+])C(=O)[O-]';  
  
# L-Histidine (protonated imidazole side chain)  
$his = 'C1=C([NH]C=[NH+]1)C[C@@H](C(=O)[O-])[NH3+]';
```

Nucleotides and nucleobases

```
# Adenine (neutral nucleobase)  
$adenine = 'C1=NC(=C2C(=N1)N=CN2)N';  
  
# Cytosine (neutral nucleobase)  
$cytosine = 'C1=CN(C(=O)N=C1N)C';
```

Peptides

```
# Dipeptide: Gly-Ala (peptide bond connects the two residues)  
$gly_ala = 'C[C@@H](C(=O)O)NC(=O)CN';
```

Use a larger canvas (480 x 320 or larger) for peptides because the extended backbone needs more horizontal space.

Troubleshooting

Canvas shows an empty box

- RDKit.js may not have loaded. Open the browser console and look for network errors.
- Confirm that the WeBWorK server allows loading external scripts from the CDN.

- Verify that `HEADER_TEXT` appears after `loadMacros` and before `BEGIN_PGML` .

Console shows "Error initializing RDKit"

- The SMILES string is probably invalid. Validate it with Python RDKit first.
- Check that the `data-smiles` attribute is properly quoted (no nested double quotes).
- Confirm that every canvas has a unique `id` .

Molecule renders but looks wrong

- Check stereochemistry: `[C@@H]` vs `[C@H]` changes the 3D arrangement.
- Check formal charges: `[NH3+]` vs `N` changes protonation.
- Check explicit hydrogens on charged atoms: `[NH+]` vs `[N+]` .
- Check ring closures: mismatched numbers produce the wrong ring system.

Best practices checklist

- Validate all SMILES with Python RDKit or an online viewer before publishing.
- Use correct formal charges, especially `[NH3+]` , `[O-]` , and `[NH+]` .
- Give every canvas a unique `id` .
- Choose canvas dimensions appropriate for the molecule complexity.
- Sort hash keys before random selection for seed stability.
- Use `[$variable]*` (with the asterisk) in PGML for HTML variables.
- Test the rendered output in a browser before assigning to students.
- Use production `mdetails` settings for student-facing problems.

Complete working example

Amino acid identification with RDKit -- A full problem that renders an amino acid structure using RDKit.js and asks students to identify it with RadioButtons.

```
## TITLE('Identify an amino acid from its structure')
## DESCRIPTION
## A chemical structure is rendered using RDKit.js. Students
## identify the amino acid from its structure.
## ENDDescription
## KEYWORDS('amino acids','RDKit','chemical structure','RadioButtons')
## DBsubject('Biochemistry')
## DBchapter('Amino Acids')
## DBsection('Structure')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

HEADER_TEXT(<<'END_HEADER');
```

```
<script src="https://unpkg.com/@rdkit/rdkit/dist/RDKit_minimal.js"></script>
<script>
let RDKitReady = null;
function getRDKit() {
  if (!RDKitReady) { RDKitReady = initRDKitModule(); }
  return RDKitReady;
}
function initMoleculeCanvases() {
  getRDKit().then(function(RDKit) {
    const canvases = document.querySelectorAll('canvas[data-smiles]');
    canvases.forEach((canvas) => {
      const smiles = canvas.dataset.smiles;
      if (!smiles) { return; }
      const mol = RDKit.get_mol(smiles);
      const mdetails = {"legend": "", "explicitMethyl": true};
      if (canvas && mol) {
        mol.draw_to_canvas_with_highlights(
          canvas, JSON.stringify(mdetails));
        mol.delete();
      }
    });
  }).catch(error => {
    console.error('Error initializing RDKit:', error);
  });
}
if (document.readyState === 'loading') {
  document.addEventListener('DOMContentLoaded', initMoleculeCanvases);
} else {
  initMoleculeCanvases();
}
</script>
END_HEADER
```

```
$smiles = 'C[C@@H](C(=O)[O-])[NH3+]';
$canvas = '<canvas id="canvas_aa" width="320" height="240" '
  . ' data-smiles="' . $smiles . '"></canvas>';
```

```
$rb = RadioButtons(
  [ 'Alanine', 'Glycine', 'Valine', 'Serine' ],
  'Alanine',
  labels => 'ABC',
  displayLabels => 1,
);
```

BEGIN_PGML

Identify the amino acid shown below.

```
[$canvas]*
```

```
[_]{ $rb}  
END_PGML
```

```
ENDDOCUMENT( );
```

Apply it today

- Copy the quick start pattern above into a new PG file and replace the SMILES string with your target molecule.
- Validate the SMILES, set the canvas size, and preview in a browser.
- For problems that compare multiple structures, give each canvas a unique ID and consider side-by-side layout.

This page titled [6.7: Rendering Chemical Structures with RDKit](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

6.8: Displaying Chemical Formulas

This page explains how to display chemical formulas and reaction equations in WeBWorK problems using the mhchem extension for MathJax. Students see properly typeset formulas such as H_2O and balanced equations without any image files or external libraries.

Use this page when: you need to display chemical formulas, ions, or reaction equations in a PGML problem. The `\ce{}` command is part of the mhchem package, which is loaded automatically by MathJax in WeBWorK. For rendering graphical molecular structures from SMILES strings, see [Rendering Chemical Structures with RDKit](#) instead.

Background on displaying chemical formulas

MathJax includes the mhchem extension, which provides the `\ce{}` command for typesetting chemical formulas and equations. This is the same syntax used in LaTeX chemistry documents and on LibreTexts pages. Because MathJax is already loaded in WeBWorK, no additional macros or JavaScript are required.

Use `\ce{}` whenever you need chemical notation in a problem. It handles subscripts, superscripts, charges, reaction arrows, and stoichiometric coefficients automatically, producing cleaner output than manually formatting with standard math mode.

Displaying chemical formulas

Inline formulas

Wrap `\ce{}` inside `\( ... \)` to display a formula inline with the surrounding text.

```
The molecular formula for glucose is \(\ce{C6H12O6}\).
```

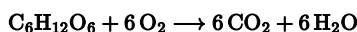
This renders as: The molecular formula for glucose is $\text{C}_6\text{H}_{12}\text{O}_6$.

Display-mode equations

Wrap `\ce{}` inside `\[ ... \]` (or use PGML double-backtick ````) to display an equation on its own centered line.

```
[``\ce{C6H12O6 + 6 O2 -> 6 CO2 + 6 H2O}``]
```

This renders as:



Common mhchem syntax

Common mhchem syntax: feature, syntax, and renders as.

Feature	Syntax	Renders as
Molecular formula	<code>\ce{H2SO4}</code>	H_2SO_4
Ionic charge	<code>\ce{SO4^2-}</code>	SO_4^{2-}
Reaction arrow	<code>\ce{A -> B}</code>	$\text{A} \longrightarrow \text{B}$
Equilibrium arrow	<code>\ce{A <=> B}</code>	$\text{A} \rightleftharpoons \text{B}$
State symbols	<code>\ce{NaCl(aq)}</code>	$\text{NaCl}(\text{aq})$
Stoichiometric coefficients	<code>\ce{2 H2 + O2 -> 2 H2O}</code>	$2 \text{H}_2 + \text{O}_2 \longrightarrow 2 \text{H}_2\text{O}$

Using `\ce{}` inside RadioButtons choices

When chemical formulas appear as answer choices in a RadioButtons widget, wrap each formula in inline math delimiters `\( \ce{...} \)`. The radio button will render the formula using MathJax.

```
$rb = RadioButtons(
    [ "\(\ce{C6H12O6} \)", "\(\ce{O2} \)", "\(\ce{CO2} \)", "\(\ce{H2O} \)" ],
    "\(\ce{C6H12O6} \)",
);
```

Complete working example

Cellular respiration with mhchem -- A full problem that displays a balanced chemical equation using `\ce{}` and asks students to identify the oxidized reactant with RadioButtons.

```
## TITLE('Identify the oxidized reactant in cellular respiration')
## DESCRIPTION
## A balanced chemical equation is rendered using mhchem. Students
## identify which reactant is being oxidized.
## ENDDescription
## KEYWORDS('mhchem','chemical formula','cellular respiration','RadioButtons')
## DBsubject('Chemistry')
## DBchapter('Chemical Reactions')
## DBsection('Oxidation-Reduction')
## Date('2026-02-26')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$rb = RadioButtons(
    [ "\(\ce{C6H12O6} \)", "\(\ce{O2} \)", "\(\ce{CO2} \)", "\(\ce{H2O} \)" ],
    "\(\ce{C6H12O6} \)",
    labels      => "ABC",
    displayLabels => 1,
    separator   => '<div style="margin-bottom: 0.7em;"></div>',
    uncheckable => 1,
    randomize   => 1,
);

BEGIN_PGML
In the following chemical reaction for cellular respiration, which reactant is being oxidized?

[`\ce{C6H12O6 + 6 O2 -> 6 CO2 + 6 H2O}`]

[_]{$rb}
```

```
END_PGML
```

```
ENDDOCUMENT ( ) ;
```

Apply it today

- Use `\ce{}` for any chemical formula or equation instead of manually formatting subscripts and superscripts in math mode.
- For inline formulas, use `\( \ce{...} \)`. For display equations, use PGML double-backtick syntax:
`[``\ce{...}``]`.
- For graphical molecular structures (ring systems, stereochemistry), use RDKit.js instead (see [Rendering Chemical Structures with RDKit](#)).

This page titled [6.8: Displaying Chemical Formulas](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

7: Testing and Debugging

This chapter helps you find and fix problems in your WeBWorK questions. Whether you are setting up local rendering for the first time, debugging a broken problem, or doing a final check before publishing, start here to find the right section.

Debugging WeBWorK problems is not fundamentally different from troubleshooting a lab protocol. You isolate the variable that changed, reproduce the failure under controlled conditions, and fix one thing at a time. The tools in this chapter give you a fast feedback loop so you spend less time guessing and more time writing good questions.

What just went wrong?

Use this table to jump straight to the section that matches your symptom. If you are not sure what went wrong, start at the top and work down.

What just went wrong?: what you see, go to, and what you will find.

What you see	Go to	What you will find
Render fails with an error message	Common Mistakes and How to Fix Them	Common mistakes organized by symptom with copy-paste fixes
Widget shows as a plain text box instead of radio buttons or dropdown	Common Mistakes and How to Fix Them	Variable scoping rules (the <code>my</code> keyword trap)
HTML tags appear as literal text in the output	Common Mistakes and How to Fix Them	The <code>[\$var]*</code> rule for HTML variables
Problem works for one seed but breaks on others	Testing Randomization and Edge Cases	Randomization testing and guard patterns
Everything looks fine but want a final checklist	QA Checklist Before Publishing	QA checklist before publishing
Need to lint or test many files at once	Scripting and Automation	Scripting and batch automation
Setting up local rendering for the first time	Setting Up the PG Renderer	Install, start, and first render walkthrough

Section map

Section map: section and what it covers.

Section	What it covers
Setting Up the PG Renderer	Install the renderer and confirm it works with your first problem file.
Common Mistakes and How to Fix Them	The big reference. Symptom-based lookup for PGML parsing, variable scoping, HTML escaping, macros, and more.
Simple Syntax Checking (Linting)	Static checks you can eyeball and renderer-based lint that catches issues before students see them.
Testing Randomization and Edge Cases	Sweep seeds to find broken variants, guard against impossible values.
QA Checklist Before Publishing	A prevention checklist and author checklist to run before you ship.
Scripting and Automation	API endpoints, Python examples, and batch linting for advanced workflows.

Start here

- If you are new to local rendering, start with [Setting Up the PG Renderer](#). It walks you through installing the renderer and confirming your first problem file works.
- If something just broke, go to [Common Mistakes and How to Fix Them](#). It is organized by symptom so you can look up what you see and find a fix without reading everything else first.
- If you are about to publish, go to [QA Checklist Before Publishing](#). The QA checklist catches the problems that look fine in author view but fail when students encounter them.

Apply it today

- Use the decision table above to jump directly to the section that matches your current situation instead of reading the chapter front to back.
- Keep the QA checklist from [QA Checklist Before Publishing](#) open as a tab while you finalize problems for a new assignment.

[7.1: Simple Syntax Checking \(Linting\) in WeBWork](#)

[7.2: Setting Up the PG Renderer](#)

[7.3: Scripting and Automation of the PG Renderer](#)

[7.4: Common Mistakes and How to Fix Them](#)

[7.5: Testing Randomization and Edge Cases](#)

[7.6: QA Checklist Before Publishing](#)

This page titled [7: Testing and Debugging](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

7.1: Simple Syntax Checking (Linting) in WeBWork

Use this section when you want to catch mistakes before students see them. Linting means checking your problem file for common errors, either by reading through it yourself (static checks) or by running it through the renderer and inspecting the output. Both approaches catch different kinds of issues, so use them together.

Think of linting like proofreading a lab protocol. Some errors you can catch by reading the text carefully. Others only show up when you actually run the protocol and watch what happens. A good QA pass uses both strategies: read the code for known pitfalls, then render the problem and inspect the result in a browser.

What linting catches that reading misses

- Variables that look correct in source but render wrong because of escaping or parsing rules (see [Introduction to PGML](#) for PGML syntax basics)
- HTML that gets escaped when you wanted it to render as formatted content
- PGML syntax that parses differently than you expected (list markers, bold, eval blocks)
- Missing macros that only fail at render time, not while reading the code (see [Common Mistakes and How to Fix Them](#) for common macro mistakes)
- Layout issues that only appear in a browser (spacing, alignment, missing blanks)

Static checks you can do by eye

These are patterns you can search for in your source file before rendering. Each one causes a specific failure that is easy to miss in the code but obvious in the rendered output.

Static checks you can do by eye reference.

What to look for	Why it matters	What to do
PGML tag wrappers inside Perl variables (<code>"[<"</code> or <code>"]{["</code> in string assignments)	PGML parses once. Tags in variables are not re-parsed.	Move PGML syntax out of variables and into the PGML block directly.
HTML in variables without <code>*</code> (variable contains <code><span</code> , <code><div</code> , etc. but is output as <code>[\$var]</code> without trailing <code>*</code>)	PGML escapes the HTML. You see literal tags instead of formatted text.	Add <code>*</code> after the bracket: <code>[\$var]*</code>
<code>MODES(</code> inside <code>[@ ... @]</code> eval blocks	MODES returns <code>1</code> in eval context, not the HTML string.	Use <code>[@ ... @]*</code> with MODES only for wrapper divs. For simple HTML, use <code>[\$var]*</code> instead.
TeX color macros (<code>\color{</code> or <code>\textcolor{</code>) in PGML	MathJax does not render these in this install.	Use HTML <code></code> styles for color instead.
Blocked HTML tags: <code><table</code> , <code><tr</code> , <code><td</code> , <code><th</code>	These tags are stripped by the HTML whitelist.	Use <code>niceTables.pl</code> (DataTable or LayoutTable) for any tables.
Ordered list label patterns: <code>A.</code> or <code>B.</code> at the start of a line in PGML	PGML may parse these as list markup instead of plain text.	Use <code>*A.*</code> (bold) or escape the dot: <code>A\.</code>

Structural checks

These checks confirm that the required scaffolding of a PG file is in place. Missing any of these causes the renderer to fail or produce partial output.

Structural checks: check, what to look for, and symptom if missing.

Check	What to look for	Symptom if missing
Missing <code>DOCUMENT()</code> ;	File does not start with <code>DOCUMENT()</code> after <code>loadMacros</code>	Render fails with a cryptic error
Missing <code>ENDDOCUMENT()</code> ;	File does not end with <code>ENDDOCUMENT()</code>	Render fails or produces partial output
Unmatched <code>BEGIN_PGML</code> / <code>END_PGML</code>	One is present without the other	Parse error, content not rendered
Nested <code>BEGIN_PGML</code> without closing	Two <code>BEGIN_PGML</code> in a row without an <code>END_PGML</code> between them	Parse error
Unbalanced <code>[@ ... @]</code> eval blocks	Opening <code>[@</code> without closing <code>@]</code> or vice versa	PGML parse failure

Renderer-based lint

Static checks catch many issues, but some problems only surface when the renderer actually processes the file. Renderer-based lint sends your file through the same pipeline students will use and reports errors, warnings, and the final HTML output.

HTML render is the primary test

WeBWorK questions are designed for browsers, so the rendered HTML is the most important check. Render your problem, save the output as an HTML file, and open it in a browser. Check: Does the question text read clearly? Are all blanks visible? Does the layout look right? Are interactive widgets (radio buttons, dropdowns) displaying correctly instead of plain text boxes?

Using the lint script

The helper script `lint_pg_via_renderer_api.py` wraps the renderer API for a local file. It sends your PG source to the renderer, collects any lint messages, and optionally saves the rendered HTML.

To get lint messages only (no HTML output), omit the `-r` flag:

```
python3 lint_pg_via_renderer_api.py -i private/myproblem.pg
```

To generate full HTML output for visual inspection, include the `-r` flag and redirect the output to a file:

```
python3 lint_pg_via_renderer_api.py -i private/myproblem.pg -r > /tmp/pg_render.html
```

The script prints the seed it uses. You can set a specific seed with `-s` :

```
python3 lint_pg_via_renderer_api.py -i private/myproblem.pg -s 1234 -r > /tmp/pg_rende
```

What to check in the output

- **Render errors:** The renderer reports line numbers and error messages. Fix these first, since they usually mean the problem will not display at all.
- **Warnings:** Check for PGML parse warnings. These may not prevent rendering but can cause unexpected behavior for some students or certain seeds.
- **Visual inspection:** Open the HTML in a browser and read it like a student. Check that text is formatted correctly, blanks appear where expected, and no raw markup is visible.
- **Widget behavior:** Confirm that interactive widgets (radio buttons, dropdowns, checkboxes) display as the correct widget type, not as a plain text input box.

A simple lint workflow

Follow these steps in order each time you finish editing a problem. The whole process takes less than a minute once you are familiar with it.

1. Save your `.pg` file.
2. Run the lint script without `-r` to check for errors.
3. Fix any reported issues.
4. Run the lint script with `-r` to generate HTML.
5. Open the HTML file in your browser.
6. Read the question as a student would.
7. Check that every answer blank appears and is in the right place.

Apply it today

Run a lint check on one problem file today. Fix any issues it reports, then open the HTML in your browser and read the question as if you were a student seeing it for the first time. If the text is clear, the blanks are visible, and the layout looks right, your problem is ready for further testing.

This page titled [7.1: Simple Syntax Checking \(Linting\) in WeBWork](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

7.2: Setting Up the PG Renderer

Use this section when you are setting up local rendering for the first time or returning after a break. The goal is to get one problem rendering in your browser so you have a fast feedback loop for editing and debugging.

Think of local rendering like setting up a bench assay. Before you run a full experiment, you confirm that your reagents work, your equipment reads correctly, and you can reproduce a known result. Here, the reagent is your `.pg` file, the equipment is the renderer container, and the known result is a cleanly rendered problem with visible answer blanks. Once this baseline works, every edit you make is a single-variable change from a known-good state.

What you need

- Podman or Docker installed on your machine
- A folder containing your `.pg` files (see [PG Problem Files](#) for PG file structure)
- A terminal (command prompt) for running commands
- A web browser for viewing rendered problems

Start the renderer

The renderer runs inside a container. Start it from your terminal with one of these commands, depending on which container tool you have installed.

1. Start the container:

```
podman compose up -d
```

or

```
docker compose up -d
```

2. Check that it is running. Open a terminal and run:

```
curl http://localhost:3000/health
```

You should see `{"status": "ok"}` in the output.

3. If you see "connection refused", the container is not running yet. Wait a few seconds and try again. The renderer can take 10 to 30 seconds to finish starting up.

What working looks like vs what broken looks like

What working looks like vs what broken looks like reference.

What you see	What it means	What to do
The <code>/health</code> endpoint returns <code>{"status": "ok"}</code>	The renderer is running and ready.	Proceed to the browser editor.
The browser editor at <code>http://localhost:3000</code> loads and rendering a simple problem shows formatted text with answer blanks	Everything is working end to end.	You are ready to start editing problems.
"Connection refused" in the terminal	The container is not running.	Wait a few seconds and try the health check again. If it still fails, check that the container started without errors.
A blank page or error page in the browser	The renderer started but something else is wrong.	Check the container logs for error messages.

Open the browser editor

1. Open `http://localhost:3000` in your browser.
2. Load a `.pg` file from the `private/` folder (see [Full file](#) for a full file example).
3. Set a seed number. Start with something simple like `1234`.
4. Click Render.

The rendered output should show formatted text, any math expressions, and answer blanks where students would type or select their answers. If you see raw PGML markup or error text instead, something needs to be fixed. [Common Mistakes and How to Fix Them](#) covers the most common causes.

Private folder workflow

Place your `.pg` files inside the `private/` folder that is mounted into the renderer container. The web editor can then load files directly using the path `private/yourfile.pg`.

Changes you save to files on your computer appear in the container automatically. There is no separate upload or sync step. Edit the file in your text editor, save it, then click Render in the browser to see the updated output.

The one rule that saves hours

When debugging, write down (1) the file path and (2) the seed. Do not change either until the failure is fixed. Change one thing at a time.

This is the same principle as a controlled experiment. If you change the seed and the code at the same time, you cannot tell which change caused the new behavior. Lock the seed, fix the code, and only move on to testing other seeds after the current one renders correctly.

What to record while you work

What to record while you work: record, example, and why it matters.

Record	Example	Why it matters
File path	<code>private/enzyme_kinetics.pg</code>	Keeps your failing case tied to one file.
Seed	<code>1234</code>	Locks the random variant for reproducibility.
What you changed	"Changed units from mM to micromolar"	Makes debugging traceable when you need to undo a change.

Your first render checklist

1. Start the renderer container.
2. Check `/health` returns `{"status":"ok"}` .
3. Open your browser to `http://localhost:3000` .
4. Load a `.pg` file from `private/` .
5. Set the seed to `1234` .
6. Click Render.
7. Read the output like a student would.
8. Confirm answer blanks appear where expected.

HTML render is the primary test

WebWork questions are designed for browsers. The rendered HTML page is the most important test of whether your problem works correctly. Open it in a browser and check that the question looks right visually: text is formatted, math displays properly, answer blanks are in the right places, and interactive widgets (radio buttons, dropdowns) appear as expected instead of plain text boxes.

Static linting and code review are useful supporting checks, but they cannot replace the rendered output. A problem that passes every lint check can still display poorly or confuse students. Always finish by reading the rendered page as a student would read it.

Apply it today

- Use local rendering for the tight edit-preview loop: edit your file, save, click Render, and check the output. This cycle should take seconds, not minutes.
- Once the local render looks right, use ADAPT for the final publish workflow. Local rendering confirms the code works; ADAPT confirms the assignment context and student view.

This page titled [7.2: Setting Up the PG Renderer](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

7.3: Scripting and Automation of the PG Renderer

Use this section when you need to render or lint problems programmatically rather than through the browser editor. Rather than writing scripts from scratch, start with the ready-made linting script below. It handles the API call, error checking, warning collection, and output formatting so you can focus on fixing problems instead of debugging your tooling.

The examples below assume the renderer is already running on `http://localhost:3000` (see [Setting Up the PG Renderer](#) for setup).

Recommended linting script

The `lint_pg_via_renderer_api.py` script is the fastest way to check a PG file from the command line. It reads a local `.pg` file, posts it to the renderer, and reports errors and warnings in one step. Copy the script below into your project or tools directory.

Quick start

```
# Lint a single file (random seed)
python3 lint_pg_via_renderer_api.py -i my_problem.pg

# Lint with a fixed seed for reproducibility
python3 lint_pg_via_renderer_api.py -i my_problem.pg -s 42

# View the rendered HTML output instead of lint messages
python3 lint_pg_via_renderer_api.py -i my_problem.pg -r

# Use a remote renderer instead of localhost
python3 lint_pg_via_renderer_api.py -i my_problem.pg -b https://renderer.example.edu
```

Command-line options

Command-line options for `lint_pg_via_renderer_api.py`.

Flag	Default	What it does
<code>-i / --input</code>	(required)	Path to the local PG file to lint
<code>-s / --seed</code>	random	Problem seed for reproducible results
<code>-b / --base-url</code>	http://localhost:3000	Renderer base URL
<code>-r / --render</code>	off	Print rendered HTML instead of lint findings
<code>-o / --output-format</code>	classic	Output format template id

What the script checks

The script collects messages from every place the renderer can report problems, so you do not need to remember which JSON field to inspect:

- `flags.error_flag` – fatal rendering errors
- `errors` and `error` – top-level error fields
- `warnings` and `warning` – top-level warning fields
- `debug.pg_warn` – PG-layer warnings (most common)
- `debug.internal` and `debug.debug` – internal renderer messages
- `renderedHTML` – scanned for embedded error blocks and "Translator errors" sections

If nothing is found in any of these fields, the script prints "No lint messages detected." and exits cleanly. If `error_flag` is set, the script raises an error so it can be used in automated pipelines where a nonzero exit code signals failure.

Batch linting

To lint every `.pg` file in a directory, combine the script with a shell loop:

```
# Lint all .pg files in a directory tree
for f in $(find problems/ -name '*.pg'); do
    echo "=== $f ==="
    python3 lint_pg_via_renderer_api.py -i "$f" -s 42
    echo
done
```

Use a fixed seed (`-s 42`) so that every failure is reproducible. If you need to sweep multiple seeds, wrap the call in a second loop over seed values.

Get the script

Download `lint_pg_via_renderer_api.py` from the [vosslab-skills GitHub repository](#) and place it in your project directory. The script uses only the Python standard library (no `pip install` required).

```
# Download the script directly
curl -O https://raw.githubusercontent.com/vosslab/vosslab-skills/main/skills/webwork-v
```

API reference

This section documents the renderer API for readers who want to understand what the script does or build custom integrations.

Endpoints

API endpoints: endpoint, method, and use for.

Endpoint	Method	Use for
<code>/render-api</code>	POST	Primary endpoint for rendering problems (used by the linting script)
<code>/</code>	POST	Alternative rendering endpoint (same behavior)
<code>/health</code>	GET	Check if the renderer is running and accepting requests

Input parameters

The renderer accepts three input types for specifying problem source. If you provide more than one, the renderer uses the highest-precedence source and ignores the rest.

Precedence order: `problemSourceURL` (highest) > `problemSource` (middle) > `sourceFilePath` (lowest)

Input parameters: parameter, what it is, and use when.

Parameter	What it is	Use when
<code>problemSource</code>	Raw PG source as a string	Source is read from a local file (this is what the linting script uses)
<code>problemSourceURL</code>	URL to a JSON endpoint that returns problem source	Problem source lives on a remote server
<code>sourceFilePath</code>	Path relative to the renderer's <code>private/</code> or <code>Library/</code> directory	File exists inside the renderer container

Response fields

The renderer returns a JSON object. Key fields:

Response fields: field, type, and description.

Field	Type	Description
<code>renderedHTML</code>	string	The rendered problem HTML (student view)
<code>flags.error_flag</code>	int	0 = success, 1 = rendering error
<code>debug.pg_warn</code>	array	Warning messages from the renderer
<code>answers</code>	object	(Instructor mode) Answer structure with correct values

Checking renderer health

Before linting or rendering, confirm the renderer is running. The `/health` endpoint returns a simple status check:

```
curl http://localhost:3000/health
```

A healthy renderer responds with a JSON object. If the request times out or returns a connection error, the container is not running. See [Setting Up the PG Renderer](#) for startup steps.

If you are scripting a pipeline, check health before submitting files:

```
import urllib.request, json

def check_health(base_url="http://localhost:3000"):
    try:
        req = urllib.request.urlopen(f"{base_url}/health", timeout=5)
        return req.status == 200
    except Exception:
        return False

if not check_health():
    print("Renderer is not running -- start the container first.")
```

Curl render example

If you prefer a quick one-off render check without Python:

```
curl -X POST "http://localhost:3000/render-api" \
  -H "Content-Type: application/json" \
  -d '{
    "problemSource": "DOCUMENT(); loadMacros(\"PGstandard.pl\", \"PGML.pl\", \"PGcourse
    "problemSeed": 42,
    "outputFormat": "classic"
  }'
```

You can also use the form-encoded style with `-F` flags if the problem source is in a file mounted inside the container:

```
curl -X POST "http://localhost:3000/render-api" \
  -F "sourceFilePath=private/myproblem.pg" \
  -F "problemSeed=1234" \
  -F "_format=json"
```

Checking for success manually

A clean render has `error_flag == 0` and an empty `pg_warn` array. Always check both, because a problem can render without a fatal error but still produce warnings. The linting script handles this automatically, but if you are writing your own integration:

```
error_flag = data.get('flags', {}).get('error_flag', 0)
warnings = data.get('debug', {}).get('pg_warn', [])

if error_flag:
    print('Rendering failed completely')
elif warnings:
    print('Rendered with warnings:')
    for w in warnings:
        print(f'  - {w}')
else:
    print('Clean render, no issues')
```

Apply it today

- Copy `lint_pg_via_renderer_api.py` into your project and lint a file with `python3 lint_pg_via_renderer_api.py -i my_problem.pg`.
- Use `-s 42` when reporting bugs so the failure is reproducible.
- Use `-r` to view the rendered HTML when you need to check formatting.
- Wrap the script in a shell loop to batch-lint an entire directory before publishing.
- The script uses only the Python standard library, so no `pip install` is needed.

This page titled [7.3: Scripting and Automation of the PG Renderer](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

7.4: Common Mistakes and How to Fix Them

Use this section when something is wrong with your WeBWorK problem and you need to find and fix it. The mistakes below are organized by what you see (the symptom), what caused it, and exactly how to fix it. Every fix includes code you can copy and paste.

These are the most common mistakes made by problem authors, collected from real debugging sessions. If your problem is not working, there is a good chance the answer is on this page.

PGML parsing errors

PGML uses asterisks and other punctuation as formatting markers (see [PGML \(PG Markup Language\)](#) for full PGML syntax). When these markers appear near variables, the parser can misinterpret them and produce confusing errors.

Asterisks around variables

Symptom: `'*' was not closed before paragraph ends`

Cause: PGML interprets `*` as the start of italic text. When you wrap a variable in single asterisks, PGML tries to treat everything between them as italic content. If the variable expands to something unexpected, the italic block never closes.

Wrong:

```
BEGIN_PGML
The answer is *[$variable]*.
END_PGML
```

Right (option 1 -- use bold instead of italic):

```
BEGIN_PGML
The answer is **[$variable]**.
END_PGML
```

Right (option 2 -- remove the emphasis):

```
BEGIN_PGML
The answer is [$variable].
END_PGML
```

Right (option 3 -- build emphasis in a variable):

```
$answer_emph = "<em>$variable</em>";

BEGIN_PGML
The answer is [$answer_emph]*.
END_PGML
```

Option 3 moves the emphasis into an HTML variable and uses `[$var]*` to pass the HTML through without escaping.

Bold markers around variables

Symptom: `'**' was not closed before paragraph ends`

Cause: Bold markers (`**`) break when the variable content contains characters that PGML interprets as additional formatting.

Wrong:

```
BEGIN_PGML
At pH [$selected_ph], histidine exists in **[$correct_label]**.
END_PGML
```

Right (remove the bold):

```
BEGIN_PGML
At pH [$selected_ph], histidine exists in [$correct_label].
END_PGML
```

Right (build emphasis in a variable):

```
$label_bold = "<strong>$correct_label</strong>";

BEGIN_PGML
At pH [$selected_ph], histidine exists in [$label_bold]*.
END_PGML
```

Using * for bullets when you also have italic content

Symptom: Confusing PGML parse behavior. Lines that begin with * and also contain **italic** markers cause the parser to misread which asterisks are bullets and which are emphasis.

Wrong:

```
BEGIN_PGML
* *State 1* (pH < 1.82): Description
* *State 2* (pH > 1.82): Description
END_PGML
```

Right (use dashes for bullets):

```
BEGIN_PGML
- **State 1** (pH < 1.82): Description
- **State 2** (pH > 1.82): Description
END_PGML
```

Variable scoping: the number one surprise

This is the single most confusing issue for new PG authors. It looks like you did everything right, but the widget does not appear.

The `my` keyword makes variables invisible to PGML

Symptom: RadioButtons renders as a plain text input box instead of radio buttons. Grading fails with Answer evaluator was not executed due to errors .

Cause: Writing `my $radio = RadioButtons(...)` creates a lexical variable that PGML cannot see. Think of it this way: PGML looks up variables in a shared directory (the package symbol table). The `my` keyword puts the variable in a private closet that PGML cannot open. When PGML encounters `[_]{$radio}` , it checks the shared directory, finds nothing, and falls back to a generic text input.

Wrong:

```
my $radio = RadioButtons(['Alpha', 'Beta'], 'Alpha');  
  
BEGIN_PGML  
[_]{$radio}  
END_PGML
```

Right:

```
$radio = RadioButtons(['Alpha', 'Beta'], 'Alpha');  
  
BEGIN_PGML  
[_]{$radio}  
END_PGML
```

The rule: Any variable referenced inside `BEGIN_PGML` (via `[$var]`, `[_]{$var}`, or `[$var]*`) must NOT use `my`. Using `my` for loop counters and temporary helpers is fine, as long as those variables never appear inside a PGML block.

```
# OK -- my for intermediates, bare for PGML-visible results  
my $idx = random(0, 3, 1);  
$rb = RadioButtons([@choices], $choices[$idx]);
```

Right:: keyword, pgml visible?, and use for.

Keyword	PGML visible?	Use for
<code>\$var = ... (bare)</code>	Yes	Variables displayed or graded in PGML
<code>local \$var = ...</code>	Yes	Temporary override of a package variable inside a block
<code>my \$var = ...</code>	No	Helper variables, loop counters, temporary computation

Why `local` works but `my` does not

Authors who know that `my` is "more modern" than `local` sometimes avoid `local` entirely. In PG, the two keywords have critically different PGML visibility. `local` temporarily modifies a package variable, so it stays in the symbol table and PGML can still find it. `my` creates a lexical variable on the Perl pad stack, which is invisible to PGML's symbol-table lookup. For PG problems, the simplest approach is to use bare package variables (no `my`, no `local`) for anything PGML needs to see.

HTML escaping: the `[$var]*` rule

Symptom: HTML tags show as literal text on screen. Instead of colored or bold text, the student sees raw markup like `<span style="color:red;">Important</span>`.

Cause: PGML escapes HTML by default for safety. Every `<` becomes `<` so the browser displays tags as text instead of rendering them.

Wrong:

```
$html_content = '<span style="color:red;">Important</span>';
```

```
BEGIN_PGML
This is [$html_content] text.
END_PGML
```

Right:

```
$html_content = '<span style="color:red;">Important</span>';

BEGIN_PGML
This is [$html_content]* text.
END_PGML
```

The rule: If a variable contains any HTML tags (`<span>` , `<div>` , `<strong>` , `<sub>` , etc.), always add `*` after the closing bracket. The asterisk tells PGML to pass the content through without escaping.

Missing or wrong macros

PG widgets like `RadioButtons` and `DropDown` live in separate macro files (see [Common PG Macros](#) for macro basics). If you forget the macro, the widget function is undefined and the renderer crashes.

Symptom table: missing macro errors

Symptom table: missing macro errors reference.

Widget or feature	Required macro	What happens without it
DropDown / PopUp	<code>parserPopUp.pl</code>	Undefined subroutine error
RadioButtons	<code>parserRadioButtons.pl</code>	Undefined subroutine error
DataTable / LayoutTable	<code>niceTables.pl</code>	Undefined subroutine error
NchooseK	<code>PGchoicemacros.pl</code>	Undefined subroutine error

How to fix

Check your `loadMacros()` block near the top of the file and add the missing macro. A correct block that covers common widgets:

```
loadMacros(
  'PGstandard.pl',
  'PGML.pl',
  'parserRadioButtons.pl',
  'parserPopUp.pl',
  'niceTables.pl',
  'PGcourse.pl',
);
```

use [statements are trapped](#)

Symptom: `'require'` trapped by operation mask

Cause: Perl's `use` statement calls `require` internally, and the PG sandbox blocks it for security. PG problems run on shared servers and cannot load arbitrary Perl modules.

Wrong:

```
use POSIX qw(floor);
use List::Util qw(shuffle);
```

Right: Use `loadMacros()` for PG macros and built-in alternatives for standard functions.

```
# For truncation (instead of POSIX floor)
$val = int(3.7);           # returns 3

# For rounding (instead of POSIX round)
$val = sprintf("%.0f", 3.7); # returns 4
```

Other trapped operations include `eval "string"` , `open()` , `system()` , and backtick execution.

Blocked HTML tags

Symptom: Table content disappears or renders as broken text.

Cause: The tags `<table>` , `<tr>` , `<td>` , and `<th>` are blocked by the HTML whitelist. The renderer strips them out silently.

Wrong:

```
BEGIN_PGML
<table>
  <tr><td>Cell 1</td><td>Cell 2</td></tr>
</table>
END_PGML
```

Right: Use `niceTables.pl` with `DataTable` or `LayoutTable` .

```
loadMacros('niceTables.pl');

$table = DataTable(
  [['Header 1', 'Header 2'],
   ['Cell 1',   'Cell 2']],
);

BEGIN_PGML
[$table]*
END_PGML
```

Note the `*` after `[$table]` because the variable contains HTML. `niceTables.pl` is the only supported way to create tables in PG problems.

MODES misuse

Symptom: The number `1` appears on screen where you expected HTML content.

Cause: `MODES(...)` returns `1` in eval context instead of the HTML string. This happens when `MODES()` is called inside a `[@ ... @]` eval block in PGML.

When MODES is needed: Only when wrapper HTML must be emitted from inside Perl eval blocks, typically for layout structures like two-column matching.

When MODES is NOT needed: Inline HTML like `<span>` , `<sub>` , and `<sup>` renders directly without `MODES()` . Store the HTML in a variable and use `[$var]*` instead.

Wrong:

```
BEGIN_PGML
[@ MODES(HTML => '<div>content</div>') @]*
END_PGML
```

Right:

```
$content = '<div>content</div>';

BEGIN_PGML
[$content]*
END_PGML
```

TeX color macros do not work. Commands like `\color{}` and `\textcolor{}` do not render in this environment. Use HTML `<span>` elements with inline CSS for colored text instead.

Order of operations

PG runs your code from top to bottom. If you reference a variable before it is assigned, it silently becomes an empty string with no warning.

Using variables before they are defined

Symptom: Empty output, missing content, or blank areas in the rendered problem.

Wrong:

```
@all_cards = ($card1, $card2, $card3);
# ... 50 lines later ...
$card1 = '<div>content 1</div>';
$card2 = '<div>content 2</div>';
$card3 = '<div>content 3</div>';
```

Right:

```
$card1 = '<div>content 1</div>';
$card2 = '<div>content 2</div>';
$card3 = '<div>content 3</div>';
# ... then use them ...
@all_cards = ($card1, $card2, $card3);
```

Shuffle timing

The correct pattern: (1) define all items, (2) create the array, (3) shuffle. Never shuffle before the items exist.

```
# 1. Define all items first
$item1 = "content 1";
$item2 = "content 2";
$item3 = "content 3";

# 2. Create array
@all_items = ($item1, $item2, $item3);

# 3. Shuffle
@shuffled = ();
@temp = @all_items;
while (@temp) {
    my $i = random(0, $#temp, 1);
    push @shuffled, splice(@temp, $i, 1);
}
```

Hash key order is not deterministic

Symptom: Random order changes between runs even with the same seed.

Cause: Perl hash key iteration order is not guaranteed. Each run can return keys in a different order, making randomization unreproducible.

Wrong:

```
%items = (a => 'Apple', b => 'Banana', c => 'Cherry');
@choices = values %items; # Random order!
```

Right:

```
%items = (a => 'Apple', b => 'Banana', c => 'Cherry');
@sorted_keys = sort keys %items;
$selected = list_random(@sorted_keys);
```

Perl-in-PG gotchas

PG is Perl, but it runs inside a security sandbox (the Safe compartment). Several standard Perl operations are blocked or behave differently.

Backslash references are broken (\@ and \%)

Symptom: Not an ARRAY reference or Not a HASH reference

Cause: The PG sandbox blocks the backslash (\) operator for creating references. Writing \@array does not produce a usable array reference.

Wrong:

```
@choices = ('Alpha', 'Beta', 'Gamma');
$rb = RadioButtons(\@choices, 'Alpha');
```

Right (option 1 -- anonymous copy, preferred):

```
@choices = ('Alpha', 'Beta', 'Gamma');  
$rb = RadioButtons([@choices], 'Alpha');
```

Right (option 2 -- PG double-tilde operator):

```
@choices = ('Alpha', 'Beta', 'Gamma');  
$rb = RadioButtons(~@choices, 'Alpha');
```

The same rule applies to hash references:

```
# Wrong:  $href = \%data;  
# Right:  $href = {%data};  
# Right:  $href = ~~%data;
```

Loops do not work inside BEGIN_TEXT

Symptom: No output, empty section where the loop content should appear.

Cause: `BEGIN_TEXT...END_TEXT` is a string construction block, not Perl code execution. A `foreach` loop inside it becomes part of the string rather than running as code.

Wrong:

```
BEGIN_TEXT  
foreach my $item (@items) {  
    \{ $item \}  
}  
END_TEXT
```

Right (build the string before BEGIN_TEXT):

```
$all_items_html = '';  
foreach my $item (@items) {  
    $all_items_html .= $item;  
}  
  
BEGIN_TEXT  
\{ $all_items_html \}  
END_TEXT
```

Answer field naming

Symptom: The answer is always marked wrong, or grading silently fails (see [Answer blanks and answers](#) for answer blank syntax).

Cause: The `ANS()` function expects an answer field created by `ans_rule()`. When you create a hidden input with JavaScript or raw HTML, its `name` attribute does not match the name that `ANS()` is looking for (e.g., `AnSwEr0001`). The answer evaluator and the input field never connect.

Wrong:

```
ANS(str_cmp($answer)); # Looks for AnSwEr0001
# But JS or HTML creates an input with a different name -- no connection!
```

Right:

```
BEGIN_TEXT
\{ MODES(HTML => '<div style="display:none;">' . ans_rule(20) . '</div>', TeX => '') \
END_TEXT
ANS(str_cmp($answer));
```

The rule: `ans_rule()` creates the input field that `ANS()` checks. Always create the `ans_rule()` first, then call `ANS()`. When you need a custom UI with JavaScript, hide the `ans_rule()` input and use JavaScript to populate its value.

RDKit and SMILES chemistry errors

If your problems use RDKit to render molecular structures from SMILES strings, incorrect SMILES notation can cause silent rendering failures or chemically wrong diagrams.

Incorrect formal charges in ring systems

Symptom: Molecule renders with wrong structure, or RDKit rejects the SMILES string entirely.

Cause: Using incorrect formal charge notation for nitrogen atoms in ring systems like imidazole. Writing NH2+ implies two hydrogens on one nitrogen, which is chemically incorrect for the imidazole ring.

Wrong:

```
# NH2+ implies TWO hydrogens on one nitrogen -- wrong for imidazole
$imidazole_protonated = '...N=C[NH2+]1...';
```

Right:

```
# Imidazole has two nitrogens: one NH, one NH+
$imidazole_protonated = 'C1=C([NH]C=[NH+]1)...';
```

Prevention: Always validate SMILES strings with Python RDKit before embedding them in PG problems. A SMILES string that parses without errors can still be chemically wrong, so check the rendered structure visually.

Debugging strategies

When the quick reference table below does not solve your problem, use these strategies to isolate the issue systematically.

Check line numbers

When variables seem empty or undefined, check where the variable is used versus where it is defined. Find the usage with a search for the variable name, then find the definition (the line with `$var_name = ...`). The definition line number must be lower (earlier in the file) than the usage line number.

Isolate PGML errors with binary search

When you get `'*' was not closed` or similar PGML parsing errors:

1. Comment out the bottom half of your PGML block.
2. Render the problem. If the error is gone, the problem is in the bottom half.
3. Restore the bottom half and comment out the top half to confirm.

4. Keep splitting the offending half until you find the exact line.
5. Check for `*` , `**` , or `_` near variables on that line.

Test with a minimal example

Create the smallest possible problem file that reproduces the error:

```
DOCUMENT();  
loadMacros("PGstandard.pl", "PGML.pl");  
  
$var = "test";  
  
BEGIN_PGML  
Test: [$var]  
END_PGML  
  
ENDDOCUMENT();
```

Add complexity one piece at a time until the problem reappears. The last piece you added is the cause.

Check renderer output

If your local setup supports the `-r` flag on the renderer script, use it to inspect the actual HTML output. Look for HTML that was escaped when it should not have been (`<` instead of `<`), missing elements, or structural problems in the generated markup.

Prevention checklist

Before committing a new PG problem file, verify each of these items:

- Variables are defined before use (check line numbers).
- No `my` on variables referenced in PGML (`$rb` , `$popup` , display variables).
- No `\@` or `\%` references. Use `@array` or `~~@array` instead.
- No `use` statements. Use `loadMacros()` for PG macros.
- No `*` or `**` immediately around `[$variables]` in PGML.
- HTML variables use `[$var]*` syntax (with the asterisk).
- No Perl loops inside `BEGIN_TEXT...END_TEXT` blocks.
- Hash keys sorted before random selection (see [Advanced Randomization Techniques](#)).
- Shuffling happens after all items are defined (see [Advanced Randomization Techniques](#)).
- `ans_rule()` created before `ANS()` call.
- SMILES strings validated with RDKit (if applicable).
- No PGML parsing errors in renderer output.

Quick reference: error message table

Use this table to look up the error or symptom you are seeing. Find your row and follow the fix in the right column.

Quick reference: error message table reference.

Error message or symptom	Likely cause	Fix
'*' was not closed	Asterisks around variable in PGML	Remove * or use ** (bold)
'***' was not closed	Bold markers around variable	Remove bold or build emphasis in a variable
Empty output / no content	Variable used before defined	Move definition before usage
HTML shown literally	Missing * in [\$var]*	Add asterisk: [\$var]*
Answer always wrong	Answer field name mismatch	Use ans_rule() before ANS()
RadioButtons renders as text input	my \$rb = RadioButtons(...)	Remove my : \$rb = RadioButtons(...)
Not an ARRAY reference	\@array backslash ref	Use [@array] or ~~@array
Not a HASH reference	\%hash backslash ref	Use {%hash} or ~~%hash
'require' trapped by operation mask	use Module in PG	Use loadMacros() or built-ins
'eval "string"' trapped	eval \$code in PG	Restructure to avoid string eval
'open' trapped by operation mask	File I/O in PG	Not possible; use PG data macros
Variable shows empty in PGML	my \$var instead of \$var	Remove my
Random order changes between runs	Hash keys not sorted	Sort keys before selection
Table content disappears	<table> tags blocked	Use niceTables.pl
Number 1 appears instead of HTML	MODES in eval context	Check MODES usage; use [\$var]* instead
A. parsed as a list item	Ordered list auto-parsing	Use *A.* or A\ to escape

Apply it today

When something breaks, come back to this page and look up the symptom. Most issues fall into one of these categories:

- A PGML parsing error caused by asterisks near variables (remove or change the markers).
- A variable that PGML cannot see because of my (remove my).
- HTML showing as literal text (add * after the bracket).
- A missing macro (add it to loadMacros()).
- A blocked HTML tag (switch to niceTables.pl).
- Variables used before they are defined (move the definition earlier).
- A backslash reference that does not work in the sandbox (use [@array]).
- Answer field name mismatches (create ans_rule() before ANS()).
- SMILES chemistry errors in RDKit problems (validate with Python RDKit).

Start with the quick reference table above. Match your error message to the row, read the fix, and apply it. If your problem is still not working, use the debugging strategies earlier on this page to isolate the issue: check line numbers, use binary search on PGML blocks, test with a minimal example, and inspect the renderer output.

This page titled [7.4: Common Mistakes and How to Fix Them](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

7.5: Testing Randomization and Edge Cases

Use this section when your problem works for one seed but you need to make sure it works for many seeds. Randomization is what makes WeBWorK problems reusable: each student gets different numbers. But different numbers can produce impossible biology, division by zero, or answers that do not make sense. Testing multiple seeds catches these problems before students encounter them.

Think of each seed as a different student sitting down with your question. You wrote the problem with one set of numbers in your head, but the randomization engine will generate hundreds of variants. Your job is to make sure every variant tells a coherent story with plausible numbers and a correct answer that makes sense.

Why randomization testing matters

- Same problem, different seed, different failure. A problem that works perfectly with seed 1234 might produce a negative concentration with seed 5678.
- Students will see many different variants. You only tested a few. The variant you did not test is the one that will confuse a student at midnight before the deadline.
- Guard patterns (described below) prevent bad values at the source, so you do not have to test every possible seed by hand.

How to sweep variants

In the browser editor, change the seed field and re-render. Try at least five to ten different seeds for any problem that uses randomization. For each variant, check the following:

- Does the story still make sense with these numbers?
- Are all numbers plausible for the biology? (No negative concentrations, no probabilities above 1, no impossible rates.)
- Does the rounding produce readable numbers? (Watch for long repeating decimals like 0.333333 that should be rounded.)
- Is the correct answer reasonable? (If the answer is 0, or negative, or astronomically large, something is wrong.)

If you are using the command-line lint script, you can render with a specific seed using the `-s` flag:

```
python3 lint_pg_via_renderer_api.py -i private/myproblem.pg -s 5678 -r > /tmp/pg_rende
```

Guard patterns: preventing bad values

Guard patterns are short code snippets that prevent randomization from producing values that would break the problem (see [Advanced Randomization Techniques](#) for additional randomization patterns and techniques). Add them in the setup section of your PG file, before the PGML block.

Avoid zero denominators

If a random variable might be zero and you later divide by it, the problem will crash for some students. Prevent this with a do/while loop or the `non_zero_random` function.

```
# Option 1: do/while loop
do { $b = random(-5, 5, 1); } while ($b == 0);

# Option 2: non_zero_random (simpler)
$b = non_zero_random(-5, 5, 1);
```

Avoid repeated values

When two variables should have different values (for example, two different enzyme concentrations to compare), use a do/while loop to ensure they are not equal.

```
$a = random(2, 9, 1);
```

```
do { $b = random(2, 9, 1); } while ($b == $a);
```

Avoid invalid domains

Some mathematical operations are undefined for certain inputs. If your problem takes the logarithm or square root of a random value, make sure that value cannot be zero or negative.

```
# Ensure $x is positive before using log($x)
do { $x = random(-5, 5, 1); } while ($x <= 0);
```

Sort hash keys before random selection

Perl hash key order is not deterministic. If you select a random key from a hash, the same seed can produce different results on different runs unless you sort the keys first.

```
@sorted_keys = sort keys %items;
$selected = list_random(@sorted_keys);
```

Shuffle after all items are defined

A common mistake is to shuffle an array before all of its elements are assigned. This produces empty or missing items in the shuffled result. Always follow this three-step pattern:

```
# Step 1: Define all items first
$item1 = "Adenine";
$item2 = "Thymine";
$item3 = "Guanine";

# Step 2: Create the array
@all_items = ($item1, $item2, $item3);

# Step 3: Shuffle (only after all items are in the array)
@temp = @all_items;
@shuffled = ();
while (@temp) {
    $index = random(0, $#temp, 1);
    push @shuffled, splice(@temp, $index, 1);
}
```

Boundary value testing

Beyond sweeping random seeds, test specific values near the boundaries of expected ranges. These are the values most likely to cause problems.

- Test values near zero, one, and other natural breakpoints in your problem.
- Test the smallest and largest values in your random range.
- Test values that produce simple fractions or exact answers to confirm rounding behaves as expected.

For biology problems, pay special attention to domain-specific constraints:

- **Concentrations:** Never negative. Usually between 0 and a realistic maximum for the context (millimolar, micromolar, etc.).
- **Probabilities:** Always between 0 and 1. Never above 1, never below 0.
- **pH values:** Between 0 and 14.

- **Rates:** Non-negative and of a reasonable magnitude for the biological process.
- **Percentages:** Between 0 and 100.

What to watch for: a checklist

What to watch for: a checklist: check, what could go wrong, and how to test.

Check	What could go wrong	How to test
Negative concentrations	Random subtraction goes below zero	Try seeds that produce small values near the lower end of the range
Impossible probabilities	Probability exceeds 1 or goes below 0	Check boundary cases where the computed probability approaches 0 or 1
Awkward rounding	Values like 0.333333... displayed without rounding	Set explicit rounding in the problem and verify the display
Repeated answer choices	Two random choices get the same value	Use a do/while guard to ensure uniqueness
Division by zero	Random denominator is zero	Use <code>non_zero_random</code> or a do/while guard
Story does not match numbers	Random values make the scenario implausible	Read each variant as a student would and ask whether the story makes sense

Apply it today

Pick one problem with randomization. Render it with five different seeds. For each variant, ask yourself: "Would this make sense to a student?" If any variant produces impossible numbers or a confusing story, add a guard pattern to prevent it. A few minutes of seed-sweeping now saves a confused email from a student later.

This page titled [7.5: Testing Randomization and Edge Cases](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

7.6: QA Checklist Before Publishing

Use this section when your problem is working and you want to make sure it is ready for students. This is the final check before publishing. The two checklists below cover code-level issues (things the computer cares about) and author-level issues (things that affect students). Run both before you ship.

These checklists are distilled from the most common problems found in real WeBWorK files (see also [Workflow and QA](#) for the full authoring workflow). Each item links back to the section where you can find a full explanation and fix. If you are not sure what an item means, follow the section reference first.

Prevention checklist: code-level checks

Walk through this table before you render for the last time. Each row is one thing that can silently break your problem without producing an obvious error message. Most of these issues will not show a helpful error -- the problem will simply render wrong or grade incorrectly.

Prevention checklist: code-level checks reference.

Check	What to verify	Section reference
Variables defined before use	All variables are assigned values before they are referenced in arrays, shuffles, or PGML	Common Mistakes and How to Fix Them
No <code>my</code> on PGML-visible variables	Variables referenced in PGML (<code>\$rb</code> , <code>\$popup</code> , display variables) do not use <code>my</code>	Common Mistakes and How to Fix Them
No <code>\@</code> or <code>\%</code> backslash references	Use <code>[@array]</code> or <code>~~@array</code> instead of <code>\@array</code>	Common Mistakes and How to Fix Them
No <code>use</code> statements	Use <code>loadMacros()</code> for PG macros, not Perl <code>use</code>	Common Mistakes and How to Fix Them
No <code>*</code> or <code>**</code> immediately around <code>[\$variables]</code> in PGML	Asterisks near variables cause parse errors such as <code>"* was not closed before paragraph ends"</code>	Common Mistakes and How to Fix Them
HTML variables use <code>[\$var]*</code> syntax	Variables containing HTML tags have the trailing asterisk so tags render instead of showing as literal text	Common Mistakes and How to Fix Them
No Perl loops inside <code>BEGIN_TEXT...END_TEXT</code> blocks	Build strings before the block, then insert them with a variable reference	Common Mistakes and How to Fix Them
Hash keys sorted before random selection	Use <code>sort keys %hash</code> before <code>list_random()</code> so a fixed seed gives stable results	Testing Randomization and Edge Cases
Shuffling happens after all items are defined	Define all items first, create the array, then shuffle	Testing Randomization and Edge Cases
OPL metadata complete	TITLE, DESCRIPTION, KEYWORDS, and DBsubject fields are filled in (see OPL Header and Metadata)	OPL Header and Metadata
No blocked HTML tags	No <code><table></code> , <code><tr></code> , <code><td></code> , <code><th></code> in PGML output (use <code>niceTables.pl</code> instead)	Common Mistakes and How to Fix Them
Required macros loaded	Every widget and feature has its macro in the <code>loadMacros()</code> call	Common Mistakes and How to Fix Them
<code>DOCUMENT()</code> and <code>ENDDOCUMENT()</code> present	File starts with <code>DOCUMENT()</code> and ends with <code>ENDDOCUMENT()</code>	Simple Syntax Checking (Linting)

Author checklist: student-facing checks

This checklist catches problems that render without errors but still confuse or frustrate students. These are things only a human can verify by reading the problem as a student would.

Author checklist: student-facing checks: check, what to verify, and how to test.

Check	What to verify	How to test
Render multiple variants	The problem works correctly with different seeds	Change seeds in the editor and render at least five variants (Testing Randomization and Edge Cases)
Displayed question matches grader	The story and the expected answer agree	Read the question and check that the correct answer matches the prompt
Correct answers accepted	The right answer is marked correct	Enter the correct answer and submit
Common wrong answers rejected	Plausible wrong answers are marked incorrect	Enter a few wrong answers (off-by-one, wrong units, sign errors) and check
Edge cases checked	Boundary values and extreme random ranges work	Test with seeds that produce small or large values (Testing Randomization and Edge Cases)
No course-local dependencies	The problem runs without files from your specific course	Test in the local renderer, not in ADAPT
Every blank has an inline evaluator	Each answer blank in PGML has a grader attached	Check that every blank has content in curly braces: <code>[_]{ans}</code> or <code>[__]{ans}</code>
Solution block included	A <code>BEGIN_PGML_SOLUTION...END_PGML_SOLUTION</code> block exists	Search for <code>SOLUTION</code> in the file
Units and rounding stated	The prompt tells students what units and precision to use	Read the question as a student and check that units and rounding are explicit

Minimal debugging note template

When you find a problem during your QA pass, fill in this template and save it. Having these details written down before you start fixing anything will save you time and prevent you from losing track of the original failure.

```
File:
Seed:
Symptom:
Error message (if any):
What I changed last:
Expected behavior:
```

Copy this block into a text file or comment in your code. When you report an issue to a colleague or come back to a problem days later, this note tells you exactly where to start.

What "ready to publish" looks like

A problem is ready to publish when all of the following are true:

- Both checklists above are satisfied with no remaining failures.
- The problem renders cleanly with at least five different seeds.
- The question reads clearly as a student-facing prompt, with no ambiguous wording or missing context.
- Every answer blank is visible and in the right location on the page.
- The solution block explains the expected method and shows the correct answer.
- No warnings or errors appear from the renderer in either HTML or JSON output.

If any of these conditions are not met, fix the issue before importing the problem to ADAPT. A problem that passes all checks here should work correctly the first time a student sees it.

Quick sanity test

If you only have time for one test, do this: render the problem with five different seeds, enter the correct answer for each, and submit. If all five accept the correct answer and the question text makes sense for each variant, the problem is probably ready. If any variant looks wrong, go back to the checklists.

Apply it today

- Pick one problem you are about to publish. Run both checklists above, row by row.
- If anything fails, fix it before importing to ADAPT. Use the section references in the code-level checklist to find the explanation and fix.
- Save the debugging note template somewhere you can find it quickly the next time something breaks.

This page titled [7.6: QA Checklist Before Publishing](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

8: Using AI Agents to Write WeBWork

AI coding agents can write WeBWork PG problem files for you, so you can focus on your subject expertise instead of learning Perl syntax. Think of it like directing a trained lab technician: you describe the experiment you want, and the technician carries out the procedure. You still need to know what a good result looks like, but you do not need to pipette every reagent yourself.

Since late 2025, several AI coding agents have become capable enough to produce working WeBWork problems from a plain-language description. This chapter walks you through choosing an agent, giving it the right reference material, connecting it to a local renderer so it can test its own output, and refining your prompts to get better problems with less back-and-forth.

What is your goal today?

Use this table to jump straight to the section that matches what you want to do. If you are not sure where to start, read the "Start here" list below the table.

Decision table for Chapter 8 sections.

What you want	Go to	What you will find
Understand what AI agents are and which ones work for WeBWork	What are AI Agents?	Overview of coding agents, comparison of current options, and what to look for when choosing one
Set up documentation so the agent writes correct PG code	Knowledge Documents for AI Agents	How to prepare and organize reference files the agent reads before generating problems
Connect the agent to a PG renderer so it can test its own output	Connecting AI Agents to the PG Renderer	Step-by-step setup for giving your agent access to a running renderer and interpreting the results
Write better prompts and refine complex problems	Advance AI Agents Advice	Prompt-engineering techniques, iterative refinement strategies, and handling edge cases

Section map

Chapter 8 section overview.

Section	What it covers
What are AI Agents?	What coding agents are, how they differ from chatbots, and a comparison of agents that can produce WeBWork problems.
Knowledge Documents for AI Agents	How to assemble and maintain the reference documents an agent needs to write correct, idiomatic PG code.
Connecting AI Agents to the PG Renderer	Giving your agent access to a local PG renderer so it can compile, render, and self-correct its output.
Advance AI Agents Advice	Prompt-engineering tips, iterative refinement workflows, and strategies for complex multi-part problems.

Start here

- If you have never used an AI coding agent, start with [What are AI Agents?](#) to understand what they are and pick one that fits your setup.
- If your agent writes code that does not render correctly, read [Knowledge Documents for AI Agents](#) to give it better reference material, then [Connecting AI Agents to the PG Renderer](#) so it can test its own output before you review it.
- If your agent already produces working problems but you want better results, go to [Advance AI Agents Advice](#) for prompt-engineering techniques and refinement strategies.

Prerequisites

- A running PG renderer on your local machine (see [Setting Up the PG Renderer](#)).
- Basic understanding of WeBWorK problem structure (see [Problem Generation \(PG\)](#)).
- An AI coding agent installed on your computer (see [What are AI Agents?](#) for options and installation links).

Apply it today

- Use the decision table above to jump directly to the section that matches your current goal instead of reading the chapter front to back.
- If this is your first time, install an agent from the list in [What are AI Agents?](#) and try asking it to write a simple multiple-choice question about your favorite color to confirm the workflow end to end.

[8.1: What are AI Agents?](#)

[8.2: Knowledge Documents for AI Agents](#)

[8.3: Connecting AI Agents to the PG Renderer](#)

[8.4: Advance AI Agents Advice](#)

This page titled [8: Using AI Agents to Write WeBWorK](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

8.1: What are AI Agents?

An AI coding agent is a program that reads your project files, runs terminal commands, and fixes its own mistakes without waiting for you to copy and paste. Unlike a chatbot, an agent works inside your development environment. You describe the biology question you want in plain English, and the agent writes the WeBWorK PG code, tests it against a local renderer, reads any error messages, and revises the code until the problem renders correctly.

This style of development is sometimes called "vibe coding," a term that gained popularity in late 2025. The idea is simple: you bring the biology expertise and a clear description of the question, and the agent brings the PG syntax and debugging. You do not need to learn Perl or memorize macro names. The agent handles those details because it has access to curated documentation and a live renderer to check its work.

Chatbots versus agents

A chatbot (such as ChatGPT in its default web interface) generates text in a conversation window. It cannot see your files, run commands, or test whether its code actually works. An AI coding agent does all of those things. The table below summarizes the key differences.

Capabilities that distinguish a chatbot from an AI coding agent.

Capability	Chatbot	AI Agent
Reads your project files	No	Yes
Runs terminal commands	No	Yes
Connects to the PG renderer	No	Yes
Self-corrects when code fails	No	Yes
Accesses curated WeBWorK documentation	No	Yes
Tests its own output	No	Yes

Why chatbots fail at WeBWorK

Even the simplest WeBWorK question defeats a standard chatbot. In a classroom demonstration, a chatbot was asked to write a problem that asks "What is your favorite color?" with a free-response answer box. The chatbot produced code that looked reasonable but failed to render. It mixed up standard Perl syntax with PG-specific conventions and could not test or fix the result.

The root cause is that PG (the Problem Generation language) looks like Perl but is not standard Perl. PG uses its own macros, its own document structure (`DOCUMENT()` through `ENDDOCUMENT()`), and its own markup language (PGML). A chatbot trained on general programming data treats PG as ordinary Perl and produces code that fails in subtle ways. When you ask the chatbot to fix the error, it often makes things worse because it cannot see the actual error message from the renderer.

This creates a debugging trap: the chatbot confidently tells you the code is correct and suggests the renderer itself must be broken. You spend time troubleshooting your Docker setup, reinstalling containers, and searching forums, when the real problem was always in the generated code. For more on common errors and how to recognize them, see [Common Mistakes and How to Fix Them](#). For renderer setup and verification, see [Setting Up the PG Renderer](#).

What makes agents work for WeBWorK

An AI coding agent succeeds where a chatbot fails because it has four capabilities that close the gap between generating code and producing a working problem.

1. **Documentation access.** The agent reads WeBWorK-specific reference documents instead of guessing from general Perl knowledge. These documents describe the correct macro signatures, PGML syntax, and problem structure. See [Knowledge Documents for AI Agents](#) for how to load these references.

2. **Renderer connection.** The agent POSTs your PG code to a local renderer (typically running at `localhost:3000`) and reads the response. This gives it immediate feedback on whether the code actually works. See [Connecting AI Agents to the PG Renderer](#) for setup details.
3. **Self-correction loop.** When the renderer returns an error, the agent reads the error message, identifies the problem, edits the code, and resubmits. This loop continues until the problem renders cleanly, without any action from you.
4. **File management.** The agent creates, edits, and organizes `.pg` files directly in your project directory. You do not need to copy and paste code between a chat window and your editor.

Available agents for WeBWork

Two commercial AI coding agents are currently well suited for writing WeBWork problems. Both run as command-line tools and can connect to a local PG renderer.

AI coding agents that support WeBWork problem development.

Agent	Provider	How it works	Documentation mechanism
Claude Code	Anthropic	Runs locally in your terminal	Skills (curated doc bundles in <code>.claude/skills/</code>)
OpenAI Codex	OpenAI	Cloud sandbox with local CLI	Forums (doc collections in the Codex panel)

Open-source alternatives such as OpenCode also exist but are not covered in this handbook.

Platform notes

- **Apple Silicon Macs.** Both agents run well as CLI tools. Claude Code and the Codex CLI are the recommended interfaces on macOS.
- **Windows.** Use WSL2 (Windows Subsystem for Linux) for the best experience with both agents and the Docker-based renderer.
- **Cloud API keys.** Both services require an API key or subscription. Keep your keys private and never commit them to a repository.
- **Cost.** Both Claude Code and Codex require paid subscriptions. Check current pricing on each provider's website.

What you need before starting

1. A running PG renderer (see [Setting Up the PG Renderer](#))
2. An AI coding agent installed (Claude Code or OpenAI Codex)
3. Knowledge documents loaded into the agent (see [Knowledge Documents for AI Agents](#))
4. A clear description of the biology problem you want to write

Apply it today

- Install Claude Code or OpenAI Codex following the provider's setup guide.
- Verify your renderer is running: open a terminal and run `curl http://localhost:3000/health` .

This page titled [8.1: What are AI Agents?](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

8.2: Knowledge Documents for AI Agents

Without WeBWorK-specific documentation, AI agents fall back on their general Perl training data and write code that looks syntactically correct but fails inside the PG sandbox. Curated reference documents prevent this. The analogy is straightforward: you would not hand a lab technician a set of reagents without a protocol and expect a reliable result. The same principle applies to AI agents. As one practitioner put it: "all the documents it needs is in the skill. So please don't search my hard drive."

This page explains what knowledge documents are, introduces a ready-to-install skill that contains them, describes what those documents cover, and shows how to build your own set if you prefer to customize.

Skills and forums: same idea, different names

Different AI agents use different terminology for the same concept: a curated collection of reference documents that the agent reads before it starts writing code.

Terminology for knowledge documents across AI agents.

Term	Used by	Where docs live
Skill	Claude Code	<code>.claude/skills/</code> directory in your project
Forum	OpenAI Codex	Forum panel in the Codex interface or a repository

The rest of this page uses "knowledge documents" as the agent-neutral term. Wherever you see it, substitute the word your agent uses.

The WeBWorK Writer skill (ready to install)

The fastest path for most users is to install a pre-built skill that already contains curated WeBWorK authoring documents. The webwork-writer skill is publicly available and maintained alongside this textbook.

Claude Code install (one command)

```
claude plugin add https://github.com/vosslab/vosslab-skills/vosslab-skills:webwork-wr
```

GitHub repository: The source documents are hosted at `https://github.com/vosslab/vosslab-skills/tree/main/skills/webwork-writer`

For Codex users: Download the reference documents from the same GitHub repository and add them to a forum in the Codex interface.

The skill contains approximately ten reference documents covering PGML structure, common mistakes, macro allowlists, renderer API usage, CSS-based color and emphasis, advanced randomization, matching problems, niceTables, linting rules, and OPL header metadata. Installing this single skill gives the agent everything it needs to write most WeBWorK problems correctly, with no manual document assembly required.

Why agents need WeBWorK-specific documents

PG looks like Perl, but it is not standard Perl. Agents trained on general Perl code will confidently use patterns that break inside the PG sandbox. Here are the most common traps:

- PG runs in a sandbox that blocks most Perl builtins. Calls to `open`, `system`, and `die` will fail silently or throw errors.
- The `my` keyword breaks radio buttons and checkboxes. PG scoping rules differ from standard Perl, and lexical variables declared with `my` become invisible to the answer evaluator.
- Backslash references like `\@array` behave differently inside PGML than in regular Perl.
- PGML is single-pass. You cannot nest PGML commands or use Perl control-flow statements (like `if` or `for`) inside a `BEGIN_PGML` block.
- Standard HTML tags are blocked in PGML. Use CSS classes injected via `HEADER_TEXT` instead.

- Colors must use CSS, not TeX `\color{}` commands. The TeX color commands are escaped before MathJax processes them.
- Only PG 2.17 macros are available. Newer Perl modules are not loaded in the renderer.

Without documentation that explains these differences, agents will hit every one of these traps. For detailed examples and fixes, see [Common Mistakes and How to Fix Them](#) and [Common PG Macros](#).

What the skill documents cover

The following table lists the topics included in the webwork-writer skill. Think of this as a table of contents for what is already inside the skill rather than a list of documents you need to assemble yourself.

Reference documents included in the webwork-writer skill.

Topic	Why the agent needs it	Textbook reference
PG authoring basics	Correct file structure and PGML syntax so every problem compiles on the first attempt	Problem Generation
Common mistakes	Avoid known PG traps like <code>my</code> scoping and backlash escaping	Common Mistakes and How to Fix Them
Macro allowlist	Only load macros that exist in PG 2.17 to avoid undefined-subroutine errors	Common PG Macros
Renderer API	POST problems to the renderer and read JSON responses for self-testing	Setting Up the PG Renderer
Color and emphasis	CSS-only styling that works inside the PG sandbox	Text Coloring and Emphasis
Randomization	Correct use of <code>random()</code> , <code>list_random()</code> , and <code>NchooseK()</code>	Advanced Randomization
niceTables	Build HTML tables inside PGML without blocked tags	niceTables
Matching problems	Randomized matching with PopUp menus or RadioButtons	Matching Problems
Lint rules	Static and renderer-based checks to catch errors before students see them	Linting
OPL header	Required metadata fields for the Open Problem Library	OPL Header and Metadata

Building your own document set

If you want to customize the reference material or use an agent that does not support skills, you can assemble your own document set. Keep these guidelines in mind:

- Keep all documents in a single directory so the agent does not search your entire filesystem. A dedicated folder like `webwork-docs/` works well.
- Include an explicit instruction in your agent configuration: "Use only the provided documents. Do not search for other files."
- Start with a minimal set and add documents only as you encounter gaps. Four documents are enough to get started: a PG authoring guide, the common mistakes list, the macro allowlist, and the renderer API reference.
- When you find a new trap or pattern, add it to your documents instead of relying on the agent to remember it from a previous conversation.

The textbook pages listed in the table above are a good source for building your own documents. You can adapt their content into plain-text or Markdown files that your agent can read.

Apply it today

- Install the webwork-writer skill with one command:

```
claude plugin add https://github.com/vosslab/vosslab-skills/vosslab-skills:webwork-
```

- Test with a simple problem. Ask your agent to write a multiple-choice question about your favorite color. If the agent produces a working PG file that renders without errors, the skill is installed correctly and the agent can find the documentation.
- Once the skill is confirmed, move on to [Connecting AI Agents to the PG Renderer](#) so the agent can compile and test its own output automatically.

This page titled [8.2: Knowledge Documents for AI Agents](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

8.3: Connecting AI Agents to the PG Renderer

The renderer connection is what separates a productive AI workflow from a chatbot copy-paste trap. Without it, you are the middleman copying code between the agent and the browser. With it, the agent tests its own work and fixes problems before you ever see them. Think of the renderer as the agent's bench assay: it runs the experiment and reads the results without asking you to hold the pipette.

Prerequisites

Before connecting your agent to the renderer, confirm that you have the following in place:

- PG renderer running on localhost:3000 (see [Setting Up the PG Renderer](#))
- Health check confirmed: `curl http://localhost:3000/health` returns a response
- Renderer API documentation included in your knowledge documents (see [Knowledge Documents for AI Agents](#))
- Agent has permission to run terminal commands (Claude Code: approve when prompted; Codex: enabled by default in sandbox)

How the agent uses the renderer

Once connected, the agent follows a feedback loop that mirrors the way you would debug a problem by hand, just faster:

1. Agent writes or edits a `.pg` file based on your prompt.
2. Agent sends the file to the renderer via HTTP POST to localhost:3000.
3. Agent reads the JSON response and checks the `error_flag` and `pg_warn` fields.
4. If errors are found, the agent reads the error messages and revises the code.
5. Agent re-sends the revised file to the renderer.
6. Steps 3 through 5 repeat until the response is clean (no errors, no warnings).
7. Agent tests the problem with multiple random seeds to catch randomization bugs.

Most simple problems take three to eight iterations. The agent handles this loop automatically. You do not need to intervene unless the agent asks a clarifying question about the biology or the intended problem behavior.

What to tell the agent about the renderer

Include the following instructions in your prompt or knowledge documents so the agent knows how to interact with the renderer:

- The renderer URL is `http://localhost:3000`.
- Always include `_format=json` in the POST request to get structured output.
- Check the `error_flag` field in the response: 0 means success, 1 means errors.
- Also check `pg_warn` for non-fatal warnings that may affect the problem.
- Use a fixed seed (for example, `problemSeed=1234`) during development for reproducible results.
- After the problem works with one seed, test with at least three different seeds.
- Run the lint tool with the `-r` flag for renderer-based lint checks (see [Linting](#)).

Typical timing

The table below shows approximate times from the video demonstration. These times are spent by the agent, not by you. While the agent works, you can review previous problems, prepare other materials, or take a break.

Expected agent time by problem complexity.

Problem complexity	Expected time
Simple (one-part, fixed choices)	3 to 5 minutes
Moderate (multi-part, basic randomization)	5 to 10 minutes
Complex (multi-part, advanced randomization, color styling)	15 to 20+ minutes

In the video, the amino acid isoelectric point question took approximately 15 minutes including the color refinement pass. This is normal for a complex problem with multiple emphasis styles and randomized data.

For batch processing of many problems, see [Scripting and Automation](#).

When the agent gets stuck

Agents occasionally hit a wall, especially with less common macros or unusual answer evaluators. When that happens, you can nudge the agent back on track without writing code yourself:

- Tell the agent to re-read its knowledge documents: "Read the skill again" or "Check the common mistakes document."
- Point the agent to a specific document when you can see what it is missing. For example: "The answer is in the RadioButtons section of the PG reference."
- Approve permission prompts promptly so the agent can continue its testing loop. A stalled permission dialog looks like a stuck agent, but the fix is just one click.
- If the agent starts searching your filesystem for example files, redirect it: "All the documents you need are in the skill. Do not search my hard drive."

For a full catalog of mistakes the agent may encounter, see [Common Mistakes](#).

Apply it today

- Confirm your agent can reach the renderer: ask it to run `curl http://localhost:3000/health` and report the result.
- Ask your agent to write and render a simple multiple-choice question and watch the feedback loop in action. A good first test: "Write a question that asks which monomer makes up cellulose, with four answer choices."

This page titled [8.3: Connecting AI Agents to the PG Renderer](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

8.4: Advance AI Agents Advice

The quality of the problem an agent produces depends on how clearly you communicate what you want. A vague prompt produces a generic question; a detailed prompt produces a polished, randomized problem ready for students. The amino acid isoelectric point problem from the video demonstrates every principle on this page.

Set boundaries before starting

Before you give the agent a biology prompt, set up a clean environment so it stays focused on your problem instead of wandering through unrelated files.

- Start with an empty project directory so the agent has a clean workspace. A folder cluttered with old drafts and unrelated code invites the agent to copy patterns that may not apply.
- Tell the agent not to search your filesystem for example files. A simple instruction like "all the documents you need are in the skill" prevents it from finding unrelated `.pg` files and copying bad patterns.
- Keep the working directory simple: one `.pg` file, the knowledge documents, and nothing else. The fewer distractions the agent has, the better it follows your instructions.
- Claude Code runs locally with full filesystem access, so setting clear boundaries matters. Codex runs in a cloud sandbox with limited access by default, but you should still keep its workspace clean.

Write detailed prompts

A good prompt tells the agent exactly what the problem should do. The table below lists the elements that produce the best results, with examples drawn from the amino acid isoelectric point problem in the video.

Prompt elements and examples from the amino acid isoelectric point video.

Prompt element	Example from video
Topic and learning objective	"amino acid isoelectric points"
Number of parts	"write a two-part question"
Interaction types	"part 1: select the charge state, part 2: calculate the pI"
Randomization requirements	"randomize the pKa values between versions so students cannot copy answers"
Visual and display preferences	"include a table showing the amino acid with its pKa values"
Constraints	"use a hypothetical amino acid with three pKa values"
Color and formatting	"color the charged groups: red for negative, blue for positive, gray for neutral"

Be patient

Complex problems take 15 to 20 minutes for the agent to write and debug. This is normal. The agent cycles through writing code, rendering it, reading error messages, and fixing problems. Each cycle brings the problem closer to working correctly.

The agent asking clarifying questions is a sign it is working carefully, not a sign of failure. Answer its questions and let it continue. Do not interrupt the agent mid-cycle. Let it finish its render-test-fix loop before giving new instructions. Breaking into the loop disrupts the agent's context and often forces it to start over.

It is better for the agent to sit there and struggle getting the problem correct than for you to sit there and struggle. You brought the biology expertise; let the agent handle the code.

Refine iteratively

Do not try to get a perfect problem on the first attempt. Use a three-pass approach that separates functionality from appearance from polish.

1. **Get it working.** The problem renders without errors and accepts correct answers. Do not worry about appearance yet. A working skeleton is always the first milestone.
2. **Improve presentation.** Add color, adjust layout, and improve readability. In the video, the instructor asked the agent to color the charge states after the basic problem was already working.
3. **Accessibility and polish.** Ensure color-blind safety (avoid red-green pairs), test with multiple random seeds, and verify all variants produce valid questions.

Accessibility and color

Color makes problems easier to read, but roughly 8 percent of males have red-green color blindness. Follow these guidelines so your problems work for everyone.

- Avoid red-green color pairs. Use high-contrast pairs instead: blue and orange, or navy and gold. The video mentions that "Bears colors" work well for contrast.
- Use gray for neutral or uncharged elements so positive and negative charges stand out clearly.
- Always pair color with another visual cue such as bold weight, font size, or a background highlight. The meaning must survive without color so that colorblind users and screen readers can still distinguish the elements.

See [Text Coloring and Emphasis](#) for CSS techniques and the recommended color palette.

Worked example: amino acid isoelectric point

The amino acid isoelectric point problem from the video illustrates the full workflow from prompt to finished problem. It also shows why detailed prompts matter and how iterative refinement produces a polished result.

The teaching moment. The instructor described a familiar classroom experience: projecting a titration curve and seeing 50 glazed eyes looking back. The curve was accurate, but students could not connect the visual to the underlying concept. This problem turns that confusing lecture slide into an interactive exercise where students work through the logic step by step.

The prompt. The instructor asked for a two-part question. Part 1: given a hypothetical amino acid with three randomized pKa values, identify which charge state has zero net charge. Part 2: calculate the isoelectric point (pI) by averaging the two pKa values that bracket the neutral state. The prompt specified randomized pKa values so that each student version would be different.

The process. The agent took approximately 15 minutes. It asked clarifying questions about the display format, created the `.pg` file, rendered it against the local PG renderer, found seed-related bugs, and fixed them automatically. The instructor did not need to touch the code at any point during this cycle.

Refinement. After the basic problem worked, the instructor asked for color. The agent added blue for positive charges and red for negative charges. A follow-up request changed neutral groups to gray so the charged groups stood out more clearly. Side chains were colored separately from backbone groups. Each refinement was a short prompt followed by another render-test-fix cycle.

The result. A fully randomized two-part problem where pKa values vary between student versions. Students identify the zero-charge state and calculate the pI, practicing the concept that used to produce glazed eyes. As the instructor put it: "I don't need to know how to program."

For techniques used in this problem, see [Advanced Randomization](#) and [Testing Randomization and Edge Cases](#).

Common pitfalls

These mistakes appear repeatedly when instructors first work with AI agents. Most are easy to prevent once you know to watch for them.

Common pitfalls when using AI agents for WeBWorK and how to prevent them.

Pitfall	What happens	Prevention
No knowledge documents	Agent writes standard Perl that fails in the PG sandbox because it does not know PG-specific conventions	Install the webwork-writer skill before starting (see Knowledge Documents for AI Agents)
Filesystem searching	Agent finds unrelated <code>.pg</code> files on your system and copies outdated or incorrect patterns	Tell the agent to use only the provided documents and keep the working directory clean
Skipping the renderer	Agent writes code it has never tested, producing problems that look plausible but fail when rendered	Connect the agent to a local renderer before starting (see Connecting AI Agents to the PG Renderer)
Not re-reading docs	Agent forgets a rule mid-session and introduces a pattern the knowledge documents explicitly warn against	Prompt the agent: "re-read the common mistakes document" when you see suspicious output
Impatient intervention	You interrupt the render-fix loop and break the agent's context, often forcing it to start over	Let the agent finish its current cycle before giving new instructions
No seed testing	Problem works for seed 1 but breaks on other seeds because a randomized value hits an edge case	Require the agent to test with at least 3 random seeds before declaring the problem done
Deprecated patterns	Agent uses legacy PG syntax that produces warnings or fails on current PG versions	Ensure knowledge documents specify PG 2.17 rules and current macro names

Scaling to many problems

Once you have one working problem, you can produce more efficiently by reusing what already works.

- Reuse a working problem as a template. Give the agent an existing `.pg` file and ask for a variation on the same topic or a parallel question for a different biological concept.
- Use batch rendering to test many files at once instead of rendering them one by one. See [Scripting and Automation](#) for how to set up batch workflows.
- Run the QA checklist on every problem before publishing, not just the first one in a set. See [QA Checklist](#) for the full list.

Apply it today

- Write a detailed prompt for one biology question following the prompt element table above. Include the topic, the number of parts, the interaction types, and what should be randomized.
- Let the agent work through the full write-render-fix cycle without interrupting. Observe how it self-corrects and how long the process takes.
- Test the finished problem with at least 5 random seeds to verify that every variant produces a valid question.
- Run the QA checklist from [QA Checklist](#) before publishing the problem to students.

This page titled [8.4: Advance AI Agents Advice](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

CHAPTER OVERVIEW

9: Appendices

This appendix collects reusable reference material: minimal templates, a concise glossary, a troubleshooting checklist, and a macro demonstration compendium with complete renderable problems for each commonly used PG macro.

Worked examples live in the main chapters where the techniques are introduced. See [Breaking Down the Components](#) for component breakdowns and [Testing and Debugging](#) for testing and debugging. Appendix 90.4 provides one complete, renderable PG problem per macro with biology-themed demos and version history notes.

[9.1: Minimal Templates](#)

[9.2: Troubleshooting Checklist](#)

[9.3: Macro Demonstrations](#)

This page titled [9: Appendices](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

9.1: Minimal Templates

This page provides copy-paste templates for common problem patterns (see [Introduction to PG Language](#) for an introduction to PG and [Breaking Down the Components](#) for detailed component breakdowns). Each template includes a complete OPL header so you can start with proper metadata.

Template: single numeric answer

```
## TITLE('Compute a simple sum')
## DESCRIPTION
## Add two values and report the total with specified units.
## ENDDescription
## KEYWORDS('addition','numeric entry','units')
## DBsubject('Biostatistics')
## DBchapter('Basic calculations')
## DBsection('Unit reporting')
## Date('2026-01-28')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "PGcourse.pl",
);

Context("Numeric");
$a = random(2, 8, 1);
$b = random(3, 9, 1);
$ans = Real($a + $b);

BEGIN_PGML
Total: [_____]{ $ans } units
END_PGML

ENDDOCUMENT();
```

Template: RadioButtons (single choice)

```
## TITLE('Select the correct label')
## DESCRIPTION
## Choose the correct label from a short list using radio buttons.
## ENDDescription
## KEYWORDS('radio buttons','single choice','labels')
## DBsubject('Cell Biology')
```

```
## DBchapter('Lab techniques')
## DBsection('Labeling conventions')
## Date('2026-01-28')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserRadioButtons.pl",
    "PGcourse.pl",
);

$radio = RadioButtons(
    [ 'Label A', 'Label B', 'Label C' ],
    'Label B',
);

BEGIN_PGML
Select the correct label: [_]{$radio}
END_PGML

ENDDOCUMENT();
```

Template: PopUp matching

```
## TITLE('Match terms to definitions')
## DESCRIPTION
## Match each term to its definition using PopUp widgets.
## ENDDescription
## KEYWORDS('matching', 'PopUp', 'definitions')
## DBsubject('Molecular Biology')
## DBchapter('Vocabulary')
## DBsection('Core terms')
## Date('2026-01-28')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "parserPopUp.pl",
    "PGcourse.pl",
);
```

```
$popup1 = PopUp(['?', 'A', 'B', 'C'], 'A');  
$popup2 = PopUp(['?', 'A', 'B', 'C'], 'B');
```

```
BEGIN_PGML
```

```
1. Term one [_]{$popup1}
```

```
2. Term two [_]{$popup2}
```

```
END_PGML
```

```
ENDDOCUMENT();
```

Template: multi-part prompt

```
## TITLE('Compute and interpret a value')  
## DESCRIPTION  
## Compute a value, then interpret it with a short follow-up statement.  
## ENDDescription  
## KEYWORDS('multi-part', 'numeric', 'interpretation')  
## DBsubject('Biochemistry')  
## DBchapter('Concentrations')  
## DBsection('Dilution')  
## Date('2026-01-28')  
## Author('Author name')  
## Institution('Institution name')
```

```
DOCUMENT();
```

```
loadMacros(  
    "PGstandard.pl",  
    "MathObjects.pl",  
    "PGML.pl",  
    "PGcourse.pl",  
);
```

```
Context("Numeric");
```

```
$c1 = random(2, 6, 1);
```

```
$c2 = random(1, 4, 1);
```

```
$ratio = Real($c1 / $c2);
```

```
$double = Real(2 * $ratio);
```

```
BEGIN_PGML
```

```
1. Compute the ratio: [_____]{$ratio}
```

```
2. What is twice the ratio? [_____]{$double}
```

```
END_PGML
```

```
ENDDOCUMENT();
```

Template: DraggableProof (ordered list)

```
## TITLE('Order a sequence of steps')
## DESCRIPTION
## Put items in the correct order using a drag-and-drop
## interface.
## ENDDescription
## KEYWORDS('ordering','DraggableProof','sequence')
## DBsubject('Molecular Biology')
## DBchapter('Techniques')
## DBsection('Protocols')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "draggableProof.pl",
    "PGcourse.pl",
);

$ordering = DraggableProof(
    [ "Step one", "Step two", "Step three" ],
    [],
    SourceLabel => "Items",
    TargetLabel => "Correct order",
);
$ordering_html = $ordering->Print;

BEGIN_PGML
Put the steps in the correct order.

[$ordering_html]*
END_PGML

ANS($ordering->cmp);

ENDDOCUMENT();
```

Template: fill-in-the-blank (String)

```
## TITLE('Name a biological term')
## DESCRIPTION
## Type the correct term in the blank using a string
## context answer.
## ENDDescription
## KEYWORDS('fill in the blank','String','terminology')
## DBsubject('Cell Biology')
## DBchapter('Vocabulary')
## DBsection('Core terms')
## Date('2026-02-15')
## Author('Author name')
## Institution('Institution name')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "contextArbitraryString.pl",
    "PGcourse.pl",
);

Context("ArbitraryString");
$ans = Compute("mitochondrion");

BEGIN_PGML
The organelle responsible for producing most of the cell's
ATP is the [_____]{$ans}.
END_PGML

ENDDOCUMENT();
```

Remember to update TITLE, DESCRIPTION, KEYWORDS, and DBsubject fields for your specific problem.

This page titled [9.1: Minimal Templates](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

9.2: Troubleshooting Checklist

This checklist maps common failure symptoms to likely causes and the quickest first fix (see [Common Mistakes and How to Fix Them](#) for detailed explanations and [Testing and Debugging](#) for the full testing and debugging guide). Use it when ADAPT reports a generic error or when grading does not match the prompt.

Common symptoms

Common symptoms: symptom, likely cause, and first fix.

Symptom	Likely cause	First fix
Generic ADAPT error	Missing macro or syntax error	Render locally and check the line number.
Blank missing or out of place	PGML formatting or wrong answer object binding	Confirm the blank token matches the answer object name.
Grading feels wrong	Tolerance or unit mismatch	Restate units next to the blank and check the checker settings.
Only some seeds make sense	Randomization range is too wide	Constrain random values and rerender multiple seeds.
CSS styling missing	HTML was escaped in PGML	Use <code>[\$var]*</code> for HTML variables.

Quick preflight

- Preview at least two seeds.
- Read the prompt as a student (units, rounding, format cues).
- Confirm each blank has the intended answer object.
- Keep a note of the file path and seed for any failure.

This page titled [9.2: Troubleshooting Checklist](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

9.3: Macro Demonstrations

This appendix provides one complete, renderable PG problem for each of the most commonly used macros documented in [Common PG Macros](#). Every demo is a full `DOCUMENT()` / `ENDDOCUMENT()` problem with a biology-themed prompt that you can copy, paste, and render immediately.

Some macros already have complete working examples elsewhere in the textbook. Where that is the case, a cross-reference replaces the demo. Each entry also includes a brief version history note so you know how long the macro has been available and what changed in recent PG releases.

Interaction widgets

[parserRadioButtons.pl](#)

History: Long-running (MathObjects era). Values display added PG 2.18 (2023).

Complete examples appear in [Multiple Choice](#) (multiple choice) and [Minimal Templates](#). See those pages for copy-paste RadioButtons demos.

[parserPopUp.pl](#)

History: Long-running (MathObjects era). DropDown method added PG 2.18. Breaking change PG 2.19.

Complete examples appear in [Matching](#) (matching) and [Minimal Templates](#). See those pages for copy-paste PopUp demos.

[PGchoicemacros.pl](#)

History: Long-running legacy macro (WeBWorK 2.0+, 2004). Predates MathObjects.

Note: PGchoicemacros.pl is a legacy macro. Its utility functions (`NchooseK` , `shuffle`) are deprecated; `PGsort` is a core function from PGbasicmacros.pl, not from this macro. The non-deprecated core is `new_match_list` with `print_q` / `print_a` . These methods generate their own form fields, so `ANS()` is required. For new problems, prefer PopUp (`parserPopUp.pl`) or RadioButtons (`parserRadioButtons.pl`).

```
## TITLE('Match organelles to functions')
## DESCRIPTION
## Use PGchoicemacros new_match_list to associate organelles
## with their primary cellular functions.
## ENDDescription
## KEYWORDS('matching','organelles','match list')
## DBsubject('Cell Biology')
## DBchapter('Organelles')
## DBsection('Functions')
## Date('2026-02-16')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGML.pl",
    "PGchoicemacros.pl",
    "PGcourse.pl",
);
```

```
$m1 = new_match_list();
$m1->qa(
    "Site of aerobic respiration",    "Mitochondrion",
    "Site of protein synthesis",      "Ribosome",
    "Packages and ships proteins",    "Golgi apparatus",
    "Contains the cell's DNA",        "Nucleus",
);
$m1->choose(4);

BEGIN_PGML
Match each function with the correct organelle.

[@ $m1->print_q @]*

[@ $m1->print_a @]*
END_PGML

ANS(str_cmp($m1->ra_correct_ans));

ENDDOCUMENT();
```

[draggableProof.pl](#)

History: Introduced ~PG 2.14. Updated PG 2.17 (2022) for Gateway quiz support.

Complete examples appear in [Ordered List](#) (ordered list) and [Minimal Templates](#). See those pages for copy-paste DraggableProof demos.

[draggableSubsets.pl](#)

History: Introduced ~PG 2.14.

```
## TITLE('Sort molecules into macromolecule categories')
## DESCRIPTION
## Drag molecules into the correct macromolecule bin using
## DraggableSubsets.
## ENDDescription
## KEYWORDS('classification','macromolecules','drag and drop')
## DBsubject('Biochemistry')
## DBchapter('Macromolecules')
## DBsection('Classification')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

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

```
"PGML.pl",
"draggableSubsets.pl",
"PGcourse.pl",
);

$draggable = DraggableSubsets(
  [ "Glucose", "Starch", "Hemoglobin", "Phospholipid",
    "Cellulose", "Insulin", "Cholesterol", "Glycogen" ],
  [
    [ 0, 1, 4, 7 ],
    [ 2, 5 ],
    [ 3, 6 ],
  ],
  DefaultSubsets => [
    { label => "Carbohydrates", indices => [] },
    { label => "Proteins", indices => [] },
    { label => "Lipids", indices => [] },
  ],
);

BEGIN_PGML
Drag each molecule into the correct macromolecule category.

[_]{$draggable}
END_PGML

ENDDOCUMENT();
```

[parserWordCompletion.pl](#)

History: Long-running (MathObjects era).

```
## TITLE('Fill in homeostasis from a word bank')
## DESCRIPTION
## Select the correct biological term from a word bank using
## WordCompletion.
## ENDDescription
## KEYWORDS('word bank','homeostasis','terminology')
## DBsubject('Physiology')
## DBchapter('Homeostasis')
## DBsection('Core concepts')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
```

```
"PGstandard.pl",
"MathObjects.pl",
"PGML.pl",
"parserWordCompletion.pl",
"PGcourse.pl",
);

Context("Numeric");
$wc = WordCompletion(
    [ "homeostasis", "metabolism", "mitosis", "osmosis", "diffusion" ],
    "homeostasis",
);

BEGIN_PGML
The process by which organisms maintain a stable internal environment
despite changes in external conditions is called
[_]{$wc}.

_word bank: homeostasis, metabolism, mitosis, osmosis, diffusion_
END_PGML

ENDDOCUMENT();
```

[PGessaymacros.pl](#)

History: Introduced ~PG 2.9 (2014). Equation editor added PG 2.16.

Note: Essay questions cannot be auto-graded. The student sees a text box and the instructor must grade manually. The `essay_box()` function creates its own form field, so `ANS(essay_cmp())` is required after `END_PGML`.

```
## TITLE('Short essay on natural selection')
## DESCRIPTION
## Prompt students to write a short explanation of natural
## selection using an essay box.
## ENDDescription
## KEYWORDS('essay','natural selection','free response')
## DBsubject('Evolution')
## DBchapter('Natural Selection')
## DBsection('Mechanisms')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "PGessaymacros.pl",
    "PGML.pl",
```

```
        "PGcourse.pl",
    );

    BEGIN_PGML
    In three to five sentences, explain how natural selection
    leads to adaptation in a population over time. Include at
    least one specific example from a real organism.

    [@ essay_box(8, 60) @]*
    END_PGML

    ANS(essay_cmp());

    ENDDOCUMENT();
```

Science-native contexts

[parserNumberWithUnits.pl](#)

History: Long-running (MathObjects era). Unit improvements PG 2.18.

```
## TITLE('Enzyme reaction rate with units')
## DESCRIPTION
## Calculate an enzyme reaction rate and report the answer
## with correct units using NumberWithUnits.
## ENDDescription
## KEYWORDS('units','enzyme kinetics','reaction rate')
## DBsubject('Biochemistry')
## DBchapter('Enzyme Kinetics')
## DBsection('Reaction rates')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "parserNumberWithUnits.pl",
    "PGcourse.pl",
);

Context("Numeric");
$moles = random(2, 8, 1);
$seconds = random(10, 30, 5);
$rate_val = $moles / $seconds;
$ans = NumberWithUnits("$rate_val mol/s");
```

```
BEGIN_PGML
```

An enzyme converts [$\$moles$] mol of substrate in [$\$seconds$] seconds.

What is the reaction rate? Include units in your answer.

Rate = [_____]{ $\$ans$ }

(Enter your answer as a number followed by units, e.g. "0.5 mol/s")

```
END_PGML
```

```
ENDDOCUMENT();
```

[parserFormulaWithUnits.pl](#)

History: Long-running (MathObjects era).

```
## TITLE('Dilution formula with units')
## DESCRIPTION
## Express a dilution formula  $C_1 \cdot V_1 / V_2$  with units using
## FormulaWithUnits.
## ENDDescription
## KEYWORDS('formula', 'dilution', 'units')
## DBsubject('Biochemistry')
## DBchapter('Solutions')
## DBsection('Dilutions')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')
```

```
DOCUMENT();
```

```
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "parserFormulaWithUnits.pl",
    "PGcourse.pl",
);
```

```
Context("Numeric");
```

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

```
 $\$ans$  = FormulaWithUnits("3 c ml");
```

```
BEGIN_PGML
```

A formula for the volume of a solution used in a serial dilution is [$V = 3c$] mL, where [c] is a concentration factor.

Express this formula with units.

[`V`] = [_____]{ $\text{\$ans}$ }

(Enter a formula with units, e.g. "3 c ml")

END_PGML

ENDDOCUMENT();

[contextScientificNotation.pl](#)

History: Long-running (MathObjects era).

```
## TITLE('Bacterial count after doublings')
## DESCRIPTION
## Calculate the number of bacteria after several doublings
## and express the answer in scientific notation.
## ENDDescription
## KEYWORDS('scientific notation','bacteria','exponential growth')
## DBsubject('Microbiology')
## DBchapter('Growth')
## DBsection('Exponential growth')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "contextScientificNotation.pl",
    "PGcourse.pl",
);

Context("ScientificNotation");
$doublings = random(10, 20, 1);
$initial = 1000;
$final = $initial * 2**$doublings;
$ans = ScientificNotation($final);

BEGIN_PGML
A bacterial culture starts with [$initial] cells and doubles
every 30 minutes. After [$doublings] doublings, how many
cells are present?
```

Express your answer in scientific notation.

Number of cells = [_____]{ $\$ans$ }

(Example format: 1.5×10^6 or 1.5E6)

END_PGML

ENDDOCUMENT();

[contextReaction.pl](#)

History: Long-running (MathObjects era). Improved PG 2.20 (2025).

```
## TITLE('Write a combustion reaction')
## DESCRIPTION
## Enter a balanced combustion reaction using the
## Reaction context.
## ENDDescription
## KEYWORDS('chemical equation','combustion','reaction')
## DBsubject('Biochemistry')
## DBchapter('Metabolism')
## DBsection('Chemical reactions')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "contextReaction.pl",
    "PGcourse.pl",
);

Context("Reaction");
$rxn = Formula("2 H_2 + O_2 --> 2 H_2O");

BEGIN_PGML
Hydrogen gas reacts with oxygen gas to produce water.

Write the balanced equation for this reaction.
Use --> for the reaction arrow. Use _ for subscripts
(e.g., H_2 for [ $\text{H}_2$ ]).

Equation: [_____]{ $\$rxn$ }
```

```
_(Example format: 2 H_2 + O_2 --> 2 H_2O)_  
END_PGML  
  
ENDDOCUMENT();
```

[contextFraction.pl](#)

History: Long-running (MathObjects era). `num / den` methods added PG 2.19 (2024). Updated PG 2.20.

```
## TITLE('Mendelian cross offspring ratio')  
## DESCRIPTION  
## Calculate the fraction of offspring with a recessive  
## phenotype from a monohybrid cross using Fraction context.  
## ENDDescription  
## KEYWORDS('fraction','Mendelian genetics','monohybrid')  
## DBsubject('Genetics')  
## DBchapter('Mendelian Genetics')  
## DBsection('Monohybrid cross')  
## Date('2026-02-15')  
## Author('Textbook demo')  
## Institution('LibreTexts')  
  
DOCUMENT();  
loadMacros(  
    "PGstandard.pl",  
    "MathObjects.pl",  
    "PGML.pl",  
    "contextFraction.pl",  
    "PGcourse.pl",  
);  
  
Context("Fraction");  
$ans = Fraction(1, 4);  
  
BEGIN_PGML  
In a monohybrid cross between two heterozygous parents  
(Aa x Aa), what fraction of the offspring are expected  
to show the recessive phenotype (aa)?  
  
Express your answer as a fraction.  
  
Fraction with recessive phenotype = [_____]{$ans}  
END_PGML  
  
ENDDOCUMENT();
```

History: Long-running (MathObjects era).

```
## TITLE('Hardy-Weinberg allele frequency as percent')
## DESCRIPTION
## Calculate a heterozygote frequency using Hardy-Weinberg
## equilibrium and express it as a percentage.
## ENDDescription
## KEYWORDS('percent','Hardy-Weinberg','allele frequency')
## DBsubject('Genetics')
## DBchapter('Population Genetics')
## DBsection('Hardy-Weinberg')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "contextPercent.pl",
    "PGcourse.pl",
);

Context("Percent");
$q_sq = list_random(0.01, 0.04, 0.09, 0.16);
$q = sqrt($q_sq);
$p = 1 - $q;
$heterozygote_freq = 2 * $p * $q;
$ans = Percent($heterozygote_freq * 100);

BEGIN_PGML
In a population at Hardy-Weinberg equilibrium, the frequency of
the homozygous recessive genotype (aa) is [$q_sq].

What is the expected frequency of heterozygotes (Aa)?

Express your answer as a percentage.

Heterozygote frequency = [_____]{$ans}

_(Include the % sign in your answer, e.g. "48%")_
END_PGML

ENDDOCUMENT();
```

[contextArbitraryString.pl](#)

History: Long-running (MathObjects era).

Complete examples appear in [Fill in the Blank](#) (fill in the blank) and [Minimal Templates](#). See those pages for copy-paste ArbitraryString demos.

Grading and feedback

[answerCustom.pl](#)

History: Long-running (MathObjects era).

```
## TITLE('Accept pH values in a custom range')
## DESCRIPTION
## Use a custom answer checker to accept any pH value
## within the physiological range as correct.
## ENDDescription
## KEYWORDS('custom checker','pH','physiology')
## DBsubject('Biochemistry')
## DBchapter('Buffers')
## DBsection('pH')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "answerCustom.pl",
    "PGcourse.pl",
);

Context("Numeric");
$correct = Real(7.4);
$checker = custom_cmp(
    $correct,
    sub {
        my ($correct_ans, $student_ans) = @_;
        return ($student_ans >= 7.35 && $student_ans <= 7.45) ? 1 : 0;
    },
);

BEGIN_PGML
Normal blood pH is tightly regulated within a narrow range.

Enter any pH value within the normal physiological range
for human blood.
```

```
pH = [_____]{$checker}

_(Hint: normal blood pH is approximately 7.35 to 7.45)_
END_PGML

ENDDOCUMENT();
```

[answerHints.pl](#)

History: Long-running. Updated PG 2.16 (2021): no hints in preview mode.

```
## TITLE('Dilution calculation with targeted error feedback')
## DESCRIPTION
## Calculate a dilution with AnswerHints providing feedback
## for common student errors.
## ENDDescription
## KEYWORDS('answer hints','dilution','feedback')
## DBsubject('Biochemistry')
## DBchapter('Solutions')
## DBsection('Dilutions')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "answerHints.pl",
    "PGcourse.pl",
);

Context("Numeric");
$c1 = 10;
$v1 = 5;
$v2 = 50;
$c2 = Real($c1 * $v1 / $v2);
$wrong_multiply = Real($c1 * $v2 / $v1);
$wrong_ratio = Real($v2 / $v1);

BEGIN_PGML
You dilute [$v1] mL of a [$c1] mM solution to a final volume
of [$v2] mL. What is the final concentration in mM?

Final concentration = [_____]{$AnswerHints(
```

```
$wrong_multiply =>
    "Check your formula:  $C_2 = C_1 \cdot V_1 / V_2$ , not  $C_1 \cdot V_2 / V_1$ .",
$wrong_ratio =>
    "You calculated the dilution factor, not the final concentration.",
$c2 => "Correct!",
)} mM
END_PGML

ENDDOCUMENT();
```

[parserMultiAnswer.pl](#)

History: Long-running (MathObjects era). Bug fix PG 2.16 for singleResult scoring.

```
## TITLE('Linked dilution blanks graded together')
## DESCRIPTION
## Enter two related dilution values that are graded
## together using MultiAnswer.
## ENDDescription
## KEYWORDS('MultiAnswer','dilution','linked grading')
## DBsubject('Biochemistry')
## DBchapter('Solutions')
## DBsection('Dilutions')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "parserMultiAnswer.pl",
    "PGcourse.pl",
);

Context("Numeric");
$vol = Real(5);
$conc = Real(2);

$ma = MultiAnswer($vol, $conc)->with(
    singleResult => 0,
    checker => sub {
        my ($correct, $student) = @_;
        my ($sv, $sc) = @{$student};
        my @scores = (0, 0);
        $scores[0] = 1 if ($sv * $sc == 10);
    }
);
```

```
$scores[1] = 1 if ($sv * $sc == 10);  
return [@scores];  
},  
);  
  
BEGIN_PGML  
You need a total of 10 mM*mL of solute. Choose a volume  
and concentration whose product equals 10.  
  
Volume (mL): [____]{$ma}  
  
Concentration (mM): [____]{$ma}  
END_PGML  
  
ENDDOCUMENT();
```

[PGgraders.pl](#)

History: Long-running (WeBWorK 2.x+). `custom_problem_grader_0_60_100` removed PG 2.20.

Note: The `install_problem_grader` call is setup code that works with PGML inline blanks. PGML blanks generate `ANS()` calls automatically, so no explicit `ANS()` is needed.

```
## TITLE('Calculate three enzyme rates with partial credit')  
## DESCRIPTION  
## Students calculate three enzyme reaction rates with  
## partial credit using full_partial_grader from PGgraders.  
## ENDDescription  
## KEYWORDS('partial credit','enzyme kinetics','grading')  
## DBsubject('Biochemistry')  
## DBchapter('Enzyme Kinetics')  
## DBsection('Reaction rates')  
## Date('2026-02-15')  
## Author('Textbook demo')  
## Institution('LibreTexts')  
  
DOCUMENT();  
loadMacros(  
    "PGstandard.pl",  
    "MathObjects.pl",  
    "PGML.pl",  
    "PGgraders.pl",  
    "PGcourse.pl",  
);  
  
install_problem_grader(~~&full_partial_grader);  
  
Context("Numeric");
```

```
$r1 = Real(10 / 2);
$r2 = Real(20 / 4);
$r3 = Real(30 / 5);

BEGIN_PGML
Calculate the reaction rate (substrate/min) for each enzyme.
You will receive partial credit for each correct answer.

Enzyme A converts 10 units in 2 min. Rate = [_____]{$r1}

Enzyme B converts 20 units in 4 min. Rate = [_____]{$r2}

Enzyme C converts 30 units in 5 min. Rate = [_____]{$r3}
END_PGML

ENDDOCUMENT();
```

[weightedGrader.pl](#)

History: Long-running (WeBWorK 2.x+).

Note: Use `weight_ans()` to wrap a checker with a weight. The wrapped checker works with PGML inline blanks.

```
## TITLE('Weighted dilution: volume then concentration')
## DESCRIPTION
## A two-part dilution problem where the concentration
## calculation is worth more, using weightedGrader.
## ENDDescription
## KEYWORDS('weighted grading','dilution','partial credit')
## DBsubject('Biochemistry')
## DBchapter('Solutions')
## DBsection('Dilutions')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
    "weightedGrader.pl",
    "PGcourse.pl",
);

install_weighted_grader();

Context("Numeric");
```

```
$dilution_factor = Real(10);
$final_conc = Real(5);

BEGIN_PGML
You dilute a 50 mM stock solution by a factor of 10.

Part 1 (30%): What is the dilution factor?
[_____]{weight_ans($dilution_factor->cmp, 30)}

Part 2 (70%): What is the final concentration in mM?
[_____]{weight_ans($final_conc->cmp, 70)}
END_PGML

ENDDOCUMENT();
```

Layout and scaffolding

[niceTables.pl](#)

History: Long-running. Overhauled PG 2.18 (2023). Auto-loaded in PGML PG 2.19.

Complete examples appear in [Making Tables with niceTables](#) (making tables with niceTables). See that page for copy-paste DataTable and LayoutTable demos.

[scaffold.pl](#)

History: Long-running. Replaces deprecated compoundProblem*.pl (deprecated PG 2.19).

Authoring warning: Each `Section` block requires its own `BEGIN_PGML` / `END_PGML` pair. Variables defined in one section are visible in later sections, but each section's PGML block must be self-contained. Rendering quirks may appear with deeply nested scaffolds.

```
## TITLE('Two-section genetics: genotype then phenotype ratio')
## DESCRIPTION
## A scaffolded problem where students first identify a
## genotype, then calculate the phenotype ratio in a
## separate section.
## ENDDescription
## KEYWORDS('scaffold','genetics','multi-section')
## DBsubject('Genetics')
## DBchapter('Mendelian Genetics')
## DBsection('Monohybrid cross')
## Date('2026-02-15')
## Author('Textbook demo')
## Institution('LibreTexts')

DOCUMENT();
loadMacros(
    "PGstandard.pl",
    "MathObjects.pl",
    "PGML.pl",
```

```
"scaffold.pl",
"parserRadioButtons.pl",
"PGcourse.pl",
);

Context("Numeric");
$genotype_radio = RadioButtons(
    [ "AA x AA", "Aa x Aa", "AA x aa", "aa x aa" ],
    "Aa x Aa",
);
$ratio = Real(3/4);

Scaffold::Begin();

Section::Begin("Part 1: Identify the cross");
BEGIN_PGML
Both parents are heterozygous for a single gene with
complete dominance. Which cross represents this mating?

[_]{$genotype_radio}
END_PGML
Section::End();

Section::Begin("Part 2: Calculate the ratio");
BEGIN_PGML
For the cross Aa x Aa, what fraction of offspring will
show the dominant phenotype?

Express your answer as a decimal.

Fraction dominant = [_____]{$ratio}
END_PGML
Section::End();

Scaffold::End();

ENDDOCUMENT();
```

[PGgraphmacros.pl](#)

History: Long-running core macro (WeBWorK 2.x+).

Complete examples appear in [Making Graphs](#) (making graphs). See that page for copy-paste graph demos with biology-themed content.

This page titled [9.3: Macro Demonstrations](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Neil R. Voss](#).

Index

1 A

absolute tolerance

[5.5: Numerical Entry](#)

accessibility

[6.1: Text Coloring and Emphasis](#)

[8.4: Advance AI Agents Advice](#)

accessible education

[InfoPage](#)

accessible tables

[3.4: Tables and Structured Data](#)

accurate

[4.6: Solution](#)

acid

[8.4: Advance AI Agents Advice](#)

ADAPT

[Licensing](#)

[5: Different Question Types in WeBWorK](#)

[5.9: Workflow and QA](#)

[Glossary](#)

adaptive homework

[InfoPage](#)

advanced PGML

[6: Advanced PGML Techniques](#)

agent

[8.3: Connecting AI Agents to the PG Renderer](#)

AI

[8.2: Knowledge Documents for AI Agents](#)

AI agent vocabulary

[Back Matter](#)

AI agents

[8: Using AI Agents to Write WeBWorK](#)

AI coding agent

[8.1: What are AI Agents?](#)

AI workflow

[8.3: Connecting AI Agents to the PG Renderer](#)

algorithmic questions

[1.3: Key Features of WeBWorK](#)

[1.4: Comparing WeBWorK to other formats](#)

algorithmic randomization

[5.1: Question Types Overview](#)

algorithmic variants

[1.2: Why use WeBWorK?](#)

allele dominance

[5.2: Multiple Choice](#)

allele frequency

[9.3: Macro Demonstrations](#)

amino

[8.4: Advance AI Agents Advice](#)

amino acids

[4.7: Putting It All Together](#)

[5.4: Matching](#)

[6.7: Rendering Chemical Structures with RDKit](#)

answer blank

[Glossary](#)

answer blanks

[3: PGML \(PG Markup Language\)](#)

answer checker

[1.5: Common Terms and Names](#)

answer checkers

[2: Problem Generation \(PG\)](#)

answer checking

[1.1: What is WeBWorK?](#)

[1.2: Why use WeBWorK?](#)

[1.3: Key Features of WeBWorK](#)

[2.1: Introduction to PG Language](#)

answer evaluation

[6.5: Randomized Matching Problems](#)

answer evaluator

[Glossary](#)

answer matching

[3.2: Answer blanks and answers](#)

answerChecking

[2.7: Future PG version features](#)

API

[7.3: Scripting and Automation of the PG Renderer](#)

[8.3: Connecting AI Agents to the PG Renderer](#)

assessment

[1: Introduction](#)

[1.2: Why use WeBWorK?](#)

[5.6: Fill in the Blank](#)

[5.7: Multipart Questions](#)

assessment systems

[5.1: Question Types Overview](#)

assessment tools

[1.4: Comparing WeBWorK to other formats](#)

assignment behavior

[5.9: Workflow and QA](#)

associations

[6.5: Randomized Matching Problems](#)

attribution

[Licensing](#)

[2.2: OPL Header and Metadata](#)

[4: Breaking Down the Components of a WeBWorK Problem](#)

Author

[4.2: OPL Header](#)

authoring

[1.1: What is WeBWorK?](#)

[1.3: Key Features of WeBWorK](#)

[3.1: Introduction to PGML](#)

authoring capabilities

[2.7: Future PG version features](#)

authoring patterns

[3: PGML \(PG Markup Language\)](#)

[6: Advanced PGML Techniques](#)

authoring terms

[1.5: Common Terms and Names](#)

automatic grading

[1.1: What is WeBWorK?](#)

automation

[7: Testing and Debugging](#)

[7.3: Scripting and Automation of the PG Renderer](#)

2 B

bacterial count

[9.3: Macro Demonstrations](#)

best practices

[8.2: Knowledge Documents for AI Agents](#)

biochemistry

[2.4: Sections within a PG question](#)

[4.7: Putting It All Together](#)

[5: Different Question Types in WeBWorK](#)

[5.3: Multiple Answer](#)

[5.4: Matching](#)

[6.7: Rendering Chemical Structures with RDKit](#)

[9.3: Macro Demonstrations](#)

biological reporting

[5.5: Numerical Entry](#)

biology

[1: Introduction](#)

[2.2: OPL Header and Metadata](#)

[3.1: Introduction to PGML](#)

[3.3: Lists and workflows](#)

[4.7: Putting It All Together](#)

[5: Different Question Types in WeBWorK](#)

[5.2: Multiple Choice](#)

[5.6: Fill in the Blank](#)

[5.9: Workflow and QA](#)

[7.5: Testing Randomization and Edge Cases](#)

[8.4: Advance AI Agents Advice](#)

biology prompts

[3.2: Answer blanks and answers](#)

biology questions

[1.1: What is WeBWorK?](#)

[8: Using AI Agents to Write WeBWorK](#)

blanks

[4.5: Statement](#)

Bloom's taxonomy

[2.2: OPL Header and Metadata](#)

browser

[7.2: Setting Up the PG Renderer](#)

buffer

[3.3: Lists and workflows](#)

3 C

calculator notation

[3.1: Introduction to PGML](#)

cell biology

[5.6: Fill in the Blank](#)

chatbot

[8.1: What are AI Agents?](#)

[8.3: Connecting AI Agents to the PG Renderer](#)

checklist

[4.6: Solution](#)

[5.9: Workflow and QA](#)

[9.2: Troubleshooting Checklist](#)

checklists

[7.6: QA Checklist Before Publishing](#)

chemical bonds

[5.4: Matching](#)

chemical formula

[6.8: Displaying Chemical Formulas](#)

chemical structures

[6.7: Rendering Chemical Structures with RDKit](#)

chemistry

[5.4: Matching](#)

[6.8: Displaying Chemical Formulas](#)

choices

[4.5: Statement](#)

clarity

- 3: PGML (PG Markup Language)
- 4: Breaking Down the Components of a WeBWorK Problem

Problem

- 4.4: Setup
- 4.5: Statement
- 4.6: Solution

clarity problem

- 3.6: Command substitution and inserts

Classification

- 2.2: OPL Header and Metadata
- 4.7: Putting It All Together

code translation

- 2.6: Legacy PG and deprecated patterns

coding

- 4.3: Preamble
- 8: Using AI Agents to Write WeBWorK
- 8.2: Knowledge Documents for AI Agents

coding mistakes

- 7.4: Common Mistakes and How to Fix Them

cognitive load

- 6.5: Randomized Matching Problems

command substitution

- 3.6: Command substitution and inserts

common failures

- 4.5: Statement

common macros

- 2: Problem Generation (PG)

common mistakes

- 3.2: Answer blanks and answers
- 7: Testing and Debugging
- 8.2: Knowledge Documents for AI Agents
- 9.2: Troubleshooting Checklist

common terms

- 1.5: Common Terms and Names

communication

- 8.4: Advance AI Agents Advice

components

- 5.3: Multiple Answer

computation

- 5.5: Numerical Entry

computed strings

- 3.6: Command substitution and inserts

concentration

- 5.7: Multipart Questions

concept checks

- 5.1: Question Types Overview

container

- 7.2: Setting Up the PG Renderer

context

- 1.5: Common Terms and Names
- 2.4: Sections within a PG question
- 2.5: Common PG Macros
- Glossary

contextUnits

- 2.7: Future PG version features

copyright

- Licensing

core concepts

- Back Matter

core question types

- 5.1: Question Types Overview

correct answer

- 4.4: Setup

Creative Commons

- Licensing

css

- 6: Advanced PGML Techniques
- 6.1: Text Coloring and Emphasis

CSS styling

- 9.2: Troubleshooting Checklist

curriculum development

- InfoPage

customization

- 6.7: Rendering Chemical Structures with RDKit
- 8.2: Knowledge Documents for AI Agents

4 D

data visualization

- 6.3: Making Graphs

DataTable

- 6.2: Making Tables with niceTables

Date

- 4.2: OPL Header

DBchapter

- 4.2: OPL Header

DBsubject

- 4.2: OPL Header

debugging

- 4.1: Full file
- 4.3: Preamble
- 4.7: Putting It All Together
- 5.9: Workflow and QA
- 7: Testing and Debugging
- 7.1: Simple Syntax Checking (Linting) in WeBWorK
- 7.2: Setting Up the PG Renderer
- 7.3: Scripting and Automation of the PG Renderer
- 7.4: Common Mistakes and How to Fix Them
- 7.6: QA Checklist Before Publishing
- 8.1: What are AI Agents?
- 8.3: Connecting AI Agents to the PG Renderer

definitions

- 6.5: Randomized Matching Problems

DESCRIPTION

- 2.2: OPL Header and Metadata
- 4.2: OPL Header

development environment

- 8.1: What are AI Agents?

diagnostic path

- 9: Appendices

digital assessments

- 1.4: Comparing WeBWorK to other formats

dilution

- 5.5: Numerical Entry
- 5.7: Multipart Questions
- 9.3: Macro Demonstrations

diploid organism

- 5.2: Multiple Choice

DNA

- 5.6: Fill in the Blank

DNA polymerase

- 5.3: Multiple Answer

dNTPs

- 5.3: Multiple Answer

Docker

- 7.2: Setting Up the PG Renderer

dynamic content

- 6: Advanced PGML Techniques

5 E

education

- 1: Introduction
- 1.3: Key Features of WeBWorK
- 1.6: Quickstart - Edit Your First Problem
- 3.1: Introduction to PGML
- 5.6: Fill in the Blank
- 8.4: Advance AI Agents Advice

education technology

- 8: Using AI Agents to Write WeBWorK

educational

- 6.1: Text Coloring and Emphasis

educational content

- 7.6: QA Checklist Before Publishing

educational material

- 6.3: Making Graphs

educational technology

- InfoPage
- 1.1: What is WeBWorK?
- 2.6: Legacy PG and deprecated patterns
- 2.7: Future PG version features
- 4.3: Preamble
- 5: Different Question Types in WeBWorK
- 6.5: Randomized Matching Problems

educational tools

- 3.4: Tables and Structured Data

emphasis

- 3.5: Layout controls

environment

- 8.4: Advance AI Agents Advice

enzyme kinetics

- 6.2: Making Tables with niceTables
- 6.3: Making Graphs

enzymes

- 2.4: Sections within a PG question

equilibrium arrow

- 6.8: Displaying Chemical Formulas

error handling

- 5.9: Workflow and QA
- 7.4: Common Mistakes and How to Fix Them

errors

- 6.1: Text Coloring and Emphasis
- 7: Testing and Debugging
- 7.1: Simple Syntax Checking (Linting) in WeBWorK
- 7.3: Scripting and Automation of the PG Renderer
- 7.4: Common Mistakes and How to Fix Them
- 7.5: Testing Randomization and Edge Cases
- 8.3: Connecting AI Agents to the PG Renderer

eukaryotic cells

- 5.2: Multiple Choice

experimental results

- 3.4: Tables and Structured Data

explanation

- 4.6: Solution

exponential growth

- 9.3: Macro Demonstrations

6 F

failure symptoms

- 9.2: Troubleshooting Checklist

feedback

- 1.2: Why use WeBWorK?
- 2.5: Common PG Macros
- 4.6: Solution

feedback loop

- 8.3: Connecting AI Agents to the PG Renderer

FIB

5.6: Fill in the Blank

fill in the blank

5.1: Question Types Overview

5.6: Fill in the Blank

final concentration

5.5: Numerical Entry

first fix

9.2: Troubleshooting Checklist

formatting

3.5: Layout controls

3.6: Command substitution and inserts

4.5: Statement

forum

8.2: Knowledge Documents for AI Agents

fractions

5.7: Multipart Questions

7.5: Testing Randomization and Edge Cases

free textbooks

InfoPage

functionality

4.3: Preamble

future features

2: Problem Generation (PG)

7 G

gel electrophoresis

5.8: Ordered List

genetics

5.2: Multiple Choice

5.6: Fill in the Blank

5.7: Multipart Questions

6.4: Advanced Randomization Techniques

9.3: Macro Demonstrations

glossary

9: Appendices

Back Matter

Glossary

grading

1.2: Why use WeBWorK?

1.3: Key Features of WeBWorK

1.6: Quickstart - Edit Your First Problem

2.1: Introduction to PG Language

2.5: Common PG Macros

3: PGML (PG Markup Language)

3.5: Layout controls

4.4: Setup

4.5: Statement

4.6: Solution

5.5: Numerical Entry

grading errors

9.2: Troubleshooting Checklist

grading logic

2.6: Legacy PG and deprecated patterns

4: Breaking Down the Components of a WeBWorK Problem

grading rubric

3.3: Lists and workflows

grading rules

3.2: Answer blanks and answers

4.1: Full file

graphs

6: Advanced PGML Techniques

6.3: Making Graphs

guidelines

6.1: Text Coloring and Emphasis

8 H

header

2.4: Sections within a PG question

headings

3.5: Layout controls

health check

7.2: Setting Up the PG Renderer

homozygous

5.7: Multipart Questions

homozygous recessive

5.2: Multiple Choice

HTML

6: Advanced PGML Techniques

6.2: Making Tables with niceTables

9.2: Troubleshooting Checklist

HTML escaping

6.5: Randomized Matching Problems

9 I

index

Back Matter

installation

8.1: What are AI Agents?

8.2: Knowledge Documents for AI Agents

Institution

4.2: OPL Header

instructional design

1: Introduction

2: Problem Generation (PG)

7.6: QA Checklist Before Publishing

8: Using AI Agents to Write WeBWorK

instructors

4.6: Solution

integration

7.3: Scripting and Automation of the PG Renderer

interaction

4.3: Preamble

interaction design

6.5: Randomized Matching Problems

interaction types

2.5: Common PG Macros

interactive content

1.4: Comparing WeBWorK to other formats

interactive learning

1.6: Quickstart - Edit Your First Problem

interactive textbooks

InfoPage

ionic charge

6.8: Displaying Chemical Formulas

issues

7.1: Simple Syntax Checking (Linting) in WeBWorK

10 J

JSON

8.3: Connecting AI Agents to the PG Renderer

11 K

key terms

6.1: Text Coloring and Emphasis

Back Matter

Glossary

KEYWORDS

2.2: OPL Header and Metadata

4.2: OPL Header

knowledge

8.2: Knowledge Documents for AI Agents

12 L

Lab techniques

4.1: Full file

lab workflow

3.2: Answer blanks and answers

Language

4.2: OPL Header

layout

3.5: Layout controls

LayoutTable

6.2: Making Tables with niceTables

learning

5.6: Fill in the Blank

6.1: Text Coloring and Emphasis

learning objective

5.3: Multiple Answer

learning outcomes

1.4: Comparing WeBWorK to other formats

legacy

2.6: Legacy PG and deprecated patterns

Level

2.2: OPL Header and Metadata

4.2: OPL Header

LibreTexts

InfoPage

Licensing

licensing

Licensing

life science

3.3: Lists and workflows

4.4: Setup

6: Advanced PGML Techniques

life sciences

1: Introduction

linting

7.1: Simple Syntax Checking (Linting) in WeBWorK

7.3: Scripting and Automation of the PG Renderer

lists

3.5: Layout controls

LMS integration

1.4: Comparing WeBWorK to other formats

local rendering

7.2: Setting Up the PG Renderer

13 M

macro

2.4: Sections within a PG question

4.3: Preamble

macro demonstration

9: Appendices

macromolecules

5.4: Matching

macros

2.5: Common PG Macros

8.1: What are AI Agents?

markup language

3.1: Introduction to PGML

Matching

5: Different Question Types in WeBWorK

5.1: Question Types Overview

math

- 1.3: Key Features of WeBWorK
- 1.6: Quickstart - Edit Your First Problem

math notation

- 3: PGML (PG Markup Language)
- 3.1: Introduction to PGML

mathematics

- 1: Introduction
- 2.7: Future PG version features

MathObjects

- 1.1: What is WeBWorK?
- 1.2: Why use WeBWorK?
- 1.5: Common Terms and Names
- 2: Problem Generation (PG)
- 2.1: Introduction to PG Language
- 2.6: Legacy PG and deprecated patterns
- Glossary

measurements

- 3.4: Tables and Structured Data
- 6.2: Making Tables with niceTables

Mendelian

- 5.7: Multipart Questions

Mendelian genetics

- 9.3: Macro Demonstrations

metadata

- 2: Problem Generation (PG)
- 2.2: OPL Header and Metadata
- 4: Breaking Down the Components of a WeBWorK Problem

- 8.2: Knowledge Documents for AI Agents

MgCl2 concentration

- 3.3: Lists and workflows

mhchem

- 6.8: Displaying Chemical Formulas

Michaelis Menten

- 6: Advanced PGML Techniques

microbiology

- 9.3: Macro Demonstrations

minimal templates

- 9: Appendices

modern matching problems

- 6.5: Randomized Matching Problems

molecular drawings

- 6.7: Rendering Chemical Structures with RDKit

molecular formula

- 6.8: Displaying Chemical Formulas

monohybrid

- 5.7: Multipart Questions

monohybrid cross

- 5.2: Multiple Choice

monomers

- 5.4: Matching

multiple answer

- 5.1: Question Types Overview
- 5.3: Multiple Answer

multiple choice

- 2.5: Common PG Macros
- 5: Different Question Types in WeBWorK
- 5.1: Question Types Overview
- 5.2: Multiple Choice
- 5.6: Fill in the Blank

14 N

narratives

- 4.5: Statement

negative concentrations

- 6.4: Advanced Randomization Techniques

niceTables

- 6.2: Making Tables with niceTables

nucleic acids

- 5.4: Matching

nucleotides

- 5.4: Matching
- 6.7: Rendering Chemical Structures with RDKit

numeric answer

- 3.2: Answer blanks and answers
- 5.5: Numerical Entry

numeric answers

- 6.3: Making Graphs

numeric practice

- 5.1: Question Types Overview

numerical entry

- 5: Different Question Types in WeBWorK
- 5.5: Numerical Entry

15 O

observations

- 3.4: Tables and Structured Data
- 6.2: Making Tables with niceTables

OER

- InfoPage

offspring

- 5.7: Multipart Questions

online assessment

- 2.1: Introduction to PG Language

online learning

- 2.7: Future PG version features
- 4.3: Preamble

open education initiatives

- InfoPage

open licensing

- Licensing

OPL

- 2.4: Sections within a PG question

OPL header

- 4: Breaking Down the Components of a WeBWorK Problem
- 4.7: Putting It All Together

ordered list

- 5.8: Ordered List

organelles

- 5.2: Multiple Choice

output

- 7.1: Simple Syntax Checking (Linting) in WeBWorK

16 P

patterns

- 6.4: Advanced Randomization Techniques

PCR

- 5.8: Ordered List

PCR reaction

- 3.3: Lists and workflows

pedagogy

- 6.2: Making Tables with niceTables

peptides

- 6.7: Rendering Chemical Structures with RDKit

percentages

- 7.5: Testing Randomization and Edge Cases

Perl

- 8.1: What are AI Agents?

Perl expression

- 3.6: Command substitution and inserts

Perl syntax

- 8: Using AI Agents to Write WeBWorK

PG

- 1.5: Common Terms and Names
- 2.1: Introduction to PG Language

PG code

- 8.1: What are AI Agents?

PG macro

- 9: Appendices

PG patterns

- 2.6: Legacy PG and deprecated patterns

PG structure

- Back Matter

PGML

- 1.5: Common Terms and Names
- 2: Problem Generation (PG)
- 2.4: Sections within a PG question
- 2.5: Common PG Macros
- 2.6: Legacy PG and deprecated patterns
- 3: PGML (PG Markup Language)
- 3.1: Introduction to PGML
- 3.2: Answer blanks and answers
- 4: Breaking Down the Components of a WeBWorK Problem
- 6.3: Making Graphs
- 7.4: Common Mistakes and How to Fix Them
- Glossary

PGML prompt

- 3.6: Command substitution and inserts

PGML syntax

- 2.7: Future PG version features

pH

- 9.3: Macro Demonstrations

pH values

- 7.5: Testing Randomization and Edge Cases

phenotype

- 5.7: Multipart Questions

platform terms

- 1.5: Common Terms and Names
- Back Matter

plotting

- 6.3: Making Graphs

Podman

- 7.2: Setting Up the PG Renderer

polysaccharides

- 5.4: Matching

PopUp widgets

- 6.5: Randomized Matching Problems

practice

- 1.2: Why use WeBWorK?

Preamble

- 4: Breaking Down the Components of a WeBWorK Problem
- 4.1: Full file
- 4.3: Preamble
- 4.7: Putting It All Together

primer concentration

- 3.3: Lists and workflows

primers

- 5.3: Multiple Answer

probability

- 7.5: Testing Randomization and Edge Cases

problem

- 4.4: Setup
- 8.4: Advance AI Agents Advice

problem authoring

- 2.5: Common PG Macros
- 2.7: Future PG version features
- 5: Different Question Types in WeBWork

problem authors

- 7.4: Common Mistakes and How to Fix Them

problem creation

- 2.6: Legacy PG and deprecated patterns

problem file

- 4.1: Full file

problem files

- 2: Problem Generation (PG)

problem generation

- 1: Introduction
 - 1.1: What is WeBWork?
- 1.6: Quickstart - Edit Your First Problem
- 8: Using AI Agents to Write WeBWork

problem libraries

- 2.6: Legacy PG and deprecated patterns

problem library

- 1.5: Common Terms and Names

problem patterns

- 9: Appendices

problem structure

- 2.1: Introduction to PG Language

problems

- 7: Testing and Debugging

programming

- 2.7: Future PG version features
- 6.3: Making Graphs
- 6.4: Advanced Randomization Techniques
- 7.4: Common Mistakes and How to Fix Them
- 8: Using AI Agents to Write WeBWork
- 8.1: What are AI Agents?

programming education

- 2.6: Legacy PG and deprecated patterns

programming language

- 2.1: Introduction to PG Language

prompt design

- 3.4: Tables and Structured Data

prompt engineering

- 8: Using AI Agents to Write WeBWork

protein synthesis

- 5.2: Multiple Choice

proteins

- 5.4: Matching

protocols

- 5.8: Ordered List

publishing readiness

- 7.6: QA Checklist Before Publishing

python

- 7.3: Scripting and Automation of the PG Renderer

17 Q

QA

- 5.9: Workflow and QA

qPCR

- 3.4: Tables and Structured Data
- 6: Advanced PGML Techniques
- 6.4: Advanced Randomization Techniques

quality

- 8.4: Advance AI Agents Advice

quality assurance

- 7.6: QA Checklist Before Publishing

question types

- 5: Different Question Types in WeBWork

quickstart

- 1.6: Quickstart - Edit Your First Problem

18 R

random selection

- 6.4: Advanced Randomization Techniques

random variables

- 6.4: Advanced Randomization Techniques

randomization

- 1.1: What is WeBWork?
- 1.3: Key Features of WeBWork
- 2.1: Introduction to PG Language
- 4.4: Setup
- 6.4: Advanced Randomization Techniques
- 7.5: Testing Randomization and Edge Cases
- 7.6: QA Checklist Before Publishing

randomness

- 9.2: Troubleshooting Checklist

rates

- 7.5: Testing Randomization and Edge Cases

rdkit

- 6.7: Rendering Chemical Structures with RDKit

reaction equations

- 6.8: Displaying Chemical Formulas

readability

- 3: PGML (PG Markup Language)
- 3.5: Layout controls
- 4.4: Setup
- 4.5: Statement

reference

- 8.2: Knowledge Documents for AI Agents

renderable problems

- 9: Appendices

renderer

- 8.3: Connecting AI Agents to the PG Renderer

rendering

- 7.1: Simple Syntax Checking (Linting) in WeBWork
- 7.2: Setting Up the PG Renderer

rendering chemical structures

- 6.8: Displaying Chemical Formulas

Reporting conventions

- 4.1: Full file

reporting rules

- 1.2: Why use WeBWork?

reproducibility

- 6.4: Advanced Randomization Techniques

responsive design

- 6.5: Randomized Matching Problems

restriction enzyme

- 5.3: Multiple Answer

revision

- 3.3: Lists and workflows

19 S

sample prep

- 5.8: Ordered List

science

- 3.4: Tables and Structured Data

science courses

- 5.8: Ordered List

science education

- 1: Introduction

scientific notation

- 9.3: Macro Demonstrations

scientific problems

- 2.1: Introduction to PG Language

script

- 7.3: Scripting and Automation of the PG Renderer

scripting

- 7: Testing and Debugging

searchability

- 4: Breaking Down the Components of a WeBWork Problem

section

- 4.1: Full file

sections

- 2: Problem Generation (PG)

sequencing

- 5.8: Ordered List

setup

- 2.4: Sections within a PG question
- 4.1: Full file
- 4.4: Setup
- 4.7: Putting It All Together
- 7.2: Setting Up the PG Renderer

Setup block

- 3.6: Command substitution and inserts

share

- Licensing

smiles

- 6.7: Rendering Chemical Structures with RDKit

software development

- 7.4: Common Mistakes and How to Fix Them

Solution

- 2.4: Sections within a PG question
- 4: Breaking Down the Components of a WeBWork Problem
- 4.1: Full file
- 4.6: Solution
- 4.7: Putting It All Together

standard PCR reaction

- 5.3: Multiple Answer

standardized quizzes

- 1.4: Comparing WeBWork to other formats

Statement

- 2.4: Sections within a PG question
- 4.1: Full file
- 4.5: Statement
- 4.7: Putting It All Together

static checks

- 7.1: Simple Syntax Checking (Linting) in WeBWork

statistics distribution functions

- 6.4: Advanced Randomization Techniques

stoichiometric coefficients

- 6.8: Displaying Chemical Formulas

structure

- 3.5: Layout controls
- 4: Breaking Down the Components of a WeBWork Problem

structured data

- 1.1: What is WeBWork?
- 3.4: Tables and Structured Data
- 6.2: Making Tables with niceTables

student engagement

- 1.6: Quickstart - Edit Your First Problem
- 5.8: Ordered List

student interaction

- 1.3: Key Features of WeBWork

students

- 4.6: Solution

styling

6.1: Text Coloring and Emphasis

syntax

7.4: Common Mistakes and How to Fix Them

syntax checking

7: Testing and Debugging

20 T

table formatting

6.2: Making Tables with niceTables

tables

3.4: Tables and Structured Data
3.5: Layout controls
3.6: Command substitution and inserts
6: Advanced PGML Techniques

tasks

3.3: Lists and workflows
3.5: Layout controls

teaching

3.1: Introduction to PGML
6.1: Text Coloring and Emphasis

teaching methodologies

5.8: Ordered List

teaching strategies

1.4: Comparing WeBWorK to other formats

teaching techniques

1: Introduction

teaching tool

6.7: Rendering Chemical Structures with RDKit

template DNA

5.3: Multiple Answer

templates

1.3: Key Features of WeBWorK
1.6: Quickstart - Edit Your First Problem

terminal

7.2: Setting Up the PG Renderer

terminology

4.5: Statement
5.1: Question Types Overview

test cases

7: Testing and Debugging

testing

5.9: Workflow and QA
7.5: Testing Randomization and Edge Cases
7.6: QA Checklist Before Publishing
8.3: Connecting AI Agents to the PG Renderer

testing and debugging

9.2: Troubleshooting Checklist

TeX

3.1: Introduction to PGML

text answers

3.2: Answer blanks and answers

textbook

4.3: Preamble
Glossary

TITLE

2.2: OPL Header and Metadata
4.2: OPL Header

tolerance

5.5: Numerical Entry

tools

2.5: Common PG Macros

troubleshooting

7: Testing and Debugging
9.2: Troubleshooting Checklist

troubleshooting checklist

9: Appendices

trust

4.6: Solution

21 U

unit conversion

5.5: Numerical Entry

user guide

2.5: Common PG Macros

user interface

4.4: Setup

22 V

validation

7.6: QA Checklist Before Publishing

values

4.4: Setup

variables

3.6: Command substitution and inserts

visual

6.1: Text Coloring and Emphasis
8.4: Advance AI Agents Advice

visual inspection

7.1: Simple Syntax Checking (Linting) in WeBWorK

vocabulary reference

Back Matter

23 W

warnings

7.3: Scripting and Automation of the PG Renderer

WeBWorK

Licensing
1.1: What is WeBWorK?
1.2: Why use WeBWorK?
1.3: Key Features of WeBWorK
1.4: Comparing WeBWorK to other formats
1.5: Common Terms and Names
2: Problem Generation (PG)
2.1: Introduction to PG Language
2.7: Future PG version features
3: PGML (PG Markup Language)
3.1: Introduction to PGML
3.2: Answer blanks and answers
4.3: Preamble
5: Different Question Types in WeBWorK
5.9: Workflow and QA
6.3: Making Graphs
7: Testing and Debugging
7.4: Common Mistakes and How to Fix Them
8: Using AI Agents to Write WeBWorK
8.1: What are AI Agents?
Glossary

widgets

Back Matter

workflow

3.3: Lists and workflows
5.9: Workflow and QA
7.1: Simple Syntax Checking (Linting) in WeBWorK

workflows

3: PGML (PG Markup Language)
5.8: Ordered List

writing

1: Introduction

This glossary defines the key terms used throughout the textbook so you can look up unfamiliar jargon quickly. For additional context on these terms, see [Common terms and names](#).

Glossary of key terms used in this textbook.

Term	Definition	Term	Definition
ADAPT	LibreTexts assessment platform where WeBWorK problems are assigned to students. See Workflow and QA for how problems move from authoring to ADAPT.	bleProof	reorder items into a correct sequence. Provided by <code>draggableProof.pl</code> . See Ordered List .
Answer blank	An input field where students type or select an answer. Created in PGML with <code>[_____]</code> syntax. See Answer blanks and answers .	Dry-run	Running a tool or script to preview changes without writing them to files. All tools in this textbook default to dry-run mode and require <code>--apply</code> to write changes.
Answer evaluator	The grading function that checks student input against the correct answer. In PGML, the evaluator is attached directly to the answer blank. See Legacy PG and deprecated patterns for old-style evaluators.	HEADER_TEXT	A PG function that injects CSS or JavaScript into the page header. Used for custom styling since standard HTML tags are blocked in PGML. See Text Coloring and Emphasis .
ArbitraryString	A MathObject that accepts free-text student input without mathematical parsing. See Macro Demonstrations for a worked example.	loadMacros	The function call in the preamble that loads macro files. Each macro unlocks specific features. See Common PG Macros .
BEGIN_PGML / END_PGML	Delimiters that mark the student-facing text block in a PG problem. Everything between them is rendered as PGML. See Introduction to PGML .	Macro	A reusable code library (<code>.pl</code> file) loaded by a problem to enable features such as widgets, grading helpers, or layout tools. See Common PG Macros .
CheckboxList	A widget that presents multiple options where students can select more than one correct answer. Provided by <code>parserCheckboxList.pl</code> . See Multiple Answer .	MathObjects	Typed answer objects (numbers, formulas, lists, strings) used for grading. They determine how student input is parsed and compared to the correct answer. See Introduction to PG Language .
Context	Rules that control what student input is allowed and how it is parsed. For example, <code>Context("Numeric")</code> accepts numbers and formulas. See Introduction to PG Language .	niceTables	A macro (<code>niceTables.pl</code>) for building formatted tables inside PGML without blocked HTML tags. See Making Tables with niceTables .
DBsubject / DBchapter	OPL header fields that classify a problem by subject area and topic. Used for searchability in the Open Problem Library. See OPL Header and Metadata .	OPL	Open Problem Library. A shared repository of WeBWorK problems with standardized metadata headers for search and discovery.
DOCUMENT() / ENDDOCUMENT()	Required opening and closing function calls in every PG problem file. <code>DOCUMENT()</code> initializes the rendering engine; <code>ENDDOCUMENT()</code> signals the end. See PG Problem Files .	OPL header	Metadata comment block at the top of a PG file used for searchability and attribution. Includes fields like TITLE, DESCRIPTION, DBsubject, and AUTHOR. See OPL Header and Metadata .
Dragga	A widget that lets students drag and	PG	The Perl-based language that defines a WeBWorK problem file. PG provides the execution sandbox, variable scoping, and grading

Term	Definition	Term	Definition
	infrastructure. See Problem Generation .		defines variables, answer objects, and randomization logic. See Setup .
PGcourse.pl	A macro loaded at the end of every <code>loadMacros</code> call. It allows course-level customization without editing individual problem files. See Common PG Macros .	Skill	A curated set of reference documents that an AI coding agent reads before writing code. Called "forums" in OpenAI Codex. See Knowledge Documents for AI Agents .
PGML	PG Markup Language. A markup layer inside PG for writing student-facing text, math, and answer blanks. See PGML (PG Markup Language) .	Solution	An optional <code>BEGIN_PGML_SOLUTION / END_PGML_SOLUTION</code> block that shows the correct answer and rationale after the due date. See Solutions .
PGstandard.pl	The core macro that loads essential PG functions. Required in every problem. See Common PG Macros .	Statement	The <code>BEGIN_PGML / END_PGML</code> block that contains the problem text, data, and answer blanks that students see. See Statement .
PopUp	A dropdown menu widget for matching and selection problems. Provided by <code>parserPopUp.pl</code> . See Matching .	WeBWorK	An open-source online homework system that delivers randomized, auto-graded math and science problems. See What is WeBWorK .
Preamble	The section of a PG file from <code>DOCUMENT()</code> through <code>loadMacros()</code> . It declares which macro libraries the problem needs. See Preamble .		
RadioButtons	A widget that presents a single-choice question with radio button options. Provided by <code>parserRadioButtons.pl</code> . See Multiple Choice .		
Renderer	The local webwork-pg-renderer service that compiles and previews PG problems outside of ADAPT. See Setting Up the PG Renderer .		
Sandbox	The restricted execution environment in which PG code runs. It blocks most Perl builtins (<code>open</code> , <code>system</code> , <code>die</code>) for security. See Common Mistakes and How to Fix Them .		
Seed	A number that reproduces one randomized variant of a problem. Each student receives a different seed, producing different random values but the same problem structure. See Testing Randomization and Edge Cases .		
Setup	The Perl code section between the preamble and the PGML block. It		

Detailed Licensing

1 Overview

Title: [The ADAPT WeBWorK Handbook](#)

Webpages: 77

All licenses found:

- [CC BY 4.0](#): 76.6% (59 pages)
- [Undeclared](#): 23.4% (18 pages)

2 By Page

- [The ADAPT WeBWorK Handbook](#) - [CC BY 4.0](#)
 - [Front Matter](#) - [CC BY 4.0](#)
 - [TitlePage](#) - [CC BY 4.0](#)
 - [InfoPage](#) - [CC BY 4.0](#)
 - [Table of Contents](#) - [CC BY 4.0](#)
 - [AI Disclaimer](#) - [Undeclared](#)
 - [Licensing](#) - [CC BY 4.0](#)
 - [1: Introduction](#) - [CC BY 4.0](#)
 - [1.1: What is WeBWorK?](#) - [CC BY 4.0](#)
 - [1.2: Why use WeBWorK?](#) - [CC BY 4.0](#)
 - [1.3: Key Features of WeBWorK](#) - [CC BY 4.0](#)
 - [1.4: Comparing WeBWorK to other formats](#) - [CC BY 4.0](#)
 - [1.5: Common Terms and Names](#) - [CC BY 4.0](#)
 - [1.6: Quickstart - Edit Your First Problem](#) - [CC BY 4.0](#)
 - [2: Problem Generation \(PG\)](#) - [CC BY 4.0](#)
 - [2.1: Introduction to PG Language](#) - [CC BY 4.0](#)
 - [2.2: OPL Header and Metadata](#) - [Undeclared](#)
 - [2.3: PG Problem Files with a .pg file extension](#) - [CC BY 4.0](#)
 - [2.4: Sections within a PG question](#) - [CC BY 4.0](#)
 - [2.5: Common PG Macros](#) - [CC BY 4.0](#)
 - [2.6: Legacy PG and deprecated patterns](#) - [CC BY 4.0](#)
 - [2.7: Future PG version features](#) - [Undeclared](#)
 - [3: PGML \(PG Markup Language\)](#) - [CC BY 4.0](#)
 - [3.1: Introduction to PGML](#) - [CC BY 4.0](#)
 - [3.2: Answer blanks and answers](#) - [CC BY 4.0](#)
 - [3.3: Lists and workflows](#) - [CC BY 4.0](#)
 - [3.4: Tables and Structured Data](#) - [CC BY 4.0](#)
 - [3.5: Layout controls](#) - [CC BY 4.0](#)
 - [3.6: Command substitution and inserts](#) - [CC BY 4.0](#)
 - [4: Breaking Down the Components of a WeBWorK Problem](#) - [CC BY 4.0](#)
 - [4.1: Full file](#) - [CC BY 4.0](#)
 - [4.2: OPL Header](#) - [Undeclared](#)
 - [4.3: Preamble](#) - [CC BY 4.0](#)
 - [4.4: Setup](#) - [CC BY 4.0](#)
 - [4.5: Statement](#) - [CC BY 4.0](#)
 - [4.6: Solution](#) - [CC BY 4.0](#)
 - [4.7: Putting It All Together](#) - [CC BY 4.0](#)
 - [5: Different Question Types in WeBWorK](#) - [CC BY 4.0](#)
 - [5.1: Question Types Overview](#) - [CC BY 4.0](#)
 - [5.2: Multiple Choice](#) - [CC BY 4.0](#)
 - [5.3: Multiple Answer](#) - [CC BY 4.0](#)
 - [5.4: Matching](#) - [CC BY 4.0](#)
 - [5.5: Numerical Entry](#) - [CC BY 4.0](#)
 - [5.6: Fill in the Blank](#) - [CC BY 4.0](#)
 - [5.7: Multipart Questions](#) - [CC BY 4.0](#)
 - [5.8: Ordered List](#) - [CC BY 4.0](#)
 - [5.9: Workflow and QA](#) - [CC BY 4.0](#)
 - [6: Advanced PGML Techniques](#) - [CC BY 4.0](#)
 - [6.1: Text Coloring and Emphasis](#) - [CC BY 4.0](#)
 - [6.2: Making Tables with niceTables](#) - [CC BY 4.0](#)
 - [6.3: Making Graphs](#) - [CC BY 4.0](#)
 - [6.4: Advanced Randomization Techniques](#) - [CC BY 4.0](#)
 - [6.5: Randomized Matching Problems](#) - [CC BY 4.0](#)
 - [6.6: Randomized MC True and False Statements](#) - [CC BY 4.0](#)
 - [6.7: Rendering Chemical Structures with RDKit](#) - [CC BY 4.0](#)
 - [6.8: Displaying Chemical Formulas](#) - [Undeclared](#)
 - [7: Testing and Debugging](#) - [CC BY 4.0](#)
 - [7.1: Simple Syntax Checking \(Linting\) in WeBWork](#) - [Undeclared](#)
 - [7.2: Setting Up the PG Renderer](#) - [CC BY 4.0](#)
 - [7.3: Scripting and Automation of the PG Renderer](#) - [Undeclared](#)
 - [7.4: Common Mistakes and How to Fix Them](#) - [CC BY 4.0](#)
 - [7.5: Testing Randomization and Edge Cases](#) - [Undeclared](#)

- 7.6: QA Checklist Before Publishing - *Undeclared*
- 8: Using AI Agents to Write WeBWorK - *Undeclared*
 - 8.1: What are AI Agents? - *CC BY 4.0*
 - 8.2: Knowledge Documents for AI Agents - *Undeclared*
 - 8.3: Connecting AI Agents to the PG Renderer - *Undeclared*
 - 8.4: Advance AI Agents Advice - *Undeclared*
- 9: Appendices - *Undeclared*
 - 9.1: Minimal Templates - *Undeclared*
 - 9.2: Troubleshooting Checklist - *Undeclared*
 - 9.3: Macro Demonstrations - *Undeclared*
- Back Matter - *CC BY 4.0*
 - Index - *CC BY 4.0*
 - Glossary - *CC BY 4.0*
 - Detailed Licensing - *CC BY 4.0*
 - Detailed Licensing - *Undeclared*

Detailed Licensing

Overview

Title: [The ADAPT WeBWorK Handbook](#)

Webpages: 77

All licenses found:

- [CC BY 4.0](#): 76.6% (59 pages)
- [Undeclared](#): 23.4% (18 pages)

By Page

- [The ADAPT WeBWorK Handbook](#) - [CC BY 4.0](#)
 - [Front Matter](#) - [CC BY 4.0](#)
 - [TitlePage](#) - [CC BY 4.0](#)
 - [InfoPage](#) - [CC BY 4.0](#)
 - [Table of Contents](#) - [CC BY 4.0](#)
 - [AI Disclaimer](#) - [Undeclared](#)
 - [Licensing](#) - [CC BY 4.0](#)
 - [1: Introduction](#) - [CC BY 4.0](#)
 - [1.1: What is WeBWorK?](#) - [CC BY 4.0](#)
 - [1.2: Why use WeBWorK?](#) - [CC BY 4.0](#)
 - [1.3: Key Features of WeBWorK](#) - [CC BY 4.0](#)
 - [1.4: Comparing WeBWorK to other formats](#) - [CC BY 4.0](#)
 - [1.5: Common Terms and Names](#) - [CC BY 4.0](#)
 - [1.6: Quickstart - Edit Your First Problem](#) - [CC BY 4.0](#)
 - [2: Problem Generation \(PG\)](#) - [CC BY 4.0](#)
 - [2.1: Introduction to PG Language](#) - [CC BY 4.0](#)
 - [2.2: OPL Header and Metadata](#) - [Undeclared](#)
 - [2.3: PG Problem Files with a .pg file extension](#) - [CC BY 4.0](#)
 - [2.4: Sections within a PG question](#) - [CC BY 4.0](#)
 - [2.5: Common PG Macros](#) - [CC BY 4.0](#)
 - [2.6: Legacy PG and deprecated patterns](#) - [CC BY 4.0](#)
 - [2.7: Future PG version features](#) - [Undeclared](#)
 - [3: PGML \(PG Markup Language\)](#) - [CC BY 4.0](#)
 - [3.1: Introduction to PGML](#) - [CC BY 4.0](#)
 - [3.2: Answer blanks and answers](#) - [CC BY 4.0](#)
 - [3.3: Lists and workflows](#) - [CC BY 4.0](#)
 - [3.4: Tables and Structured Data](#) - [CC BY 4.0](#)
 - [3.5: Layout controls](#) - [CC BY 4.0](#)
 - [3.6: Command substitution and inserts](#) - [CC BY 4.0](#)
 - [4: Breaking Down the Components of a WeBWorK Problem](#) - [CC BY 4.0](#)
 - [4.1: Full file](#) - [CC BY 4.0](#)
 - [4.2: OPL Header](#) - [Undeclared](#)
 - [4.3: Preamble](#) - [CC BY 4.0](#)
 - [4.4: Setup](#) - [CC BY 4.0](#)
 - [4.5: Statement](#) - [CC BY 4.0](#)
 - [4.6: Solution](#) - [CC BY 4.0](#)
 - [4.7: Putting It All Together](#) - [CC BY 4.0](#)
 - [5: Different Question Types in WeBWorK](#) - [CC BY 4.0](#)
 - [5.1: Question Types Overview](#) - [CC BY 4.0](#)
 - [5.2: Multiple Choice](#) - [CC BY 4.0](#)
 - [5.3: Multiple Answer](#) - [CC BY 4.0](#)
 - [5.4: Matching](#) - [CC BY 4.0](#)
 - [5.5: Numerical Entry](#) - [CC BY 4.0](#)
 - [5.6: Fill in the Blank](#) - [CC BY 4.0](#)
 - [5.7: Multipart Questions](#) - [CC BY 4.0](#)
 - [5.8: Ordered List](#) - [CC BY 4.0](#)
 - [5.9: Workflow and QA](#) - [CC BY 4.0](#)
 - [6: Advanced PGML Techniques](#) - [CC BY 4.0](#)
 - [6.1: Text Coloring and Emphasis](#) - [CC BY 4.0](#)
 - [6.2: Making Tables with niceTables](#) - [CC BY 4.0](#)
 - [6.3: Making Graphs](#) - [CC BY 4.0](#)
 - [6.4: Advanced Randomization Techniques](#) - [CC BY 4.0](#)
 - [6.5: Randomized Matching Problems](#) - [CC BY 4.0](#)
 - [6.6: Randomized MC True and False Statements](#) - [CC BY 4.0](#)
 - [6.7: Rendering Chemical Structures with RDKit](#) - [CC BY 4.0](#)
 - [6.8: Displaying Chemical Formulas](#) - [Undeclared](#)
 - [7: Testing and Debugging](#) - [CC BY 4.0](#)
 - [7.1: Simple Syntax Checking \(Linting\) in WeBWork](#) - [Undeclared](#)
 - [7.2: Setting Up the PG Renderer](#) - [CC BY 4.0](#)
 - [7.3: Scripting and Automation of the PG Renderer](#) - [Undeclared](#)
 - [7.4: Common Mistakes and How to Fix Them](#) - [CC BY 4.0](#)
 - [7.5: Testing Randomization and Edge Cases](#) - [Undeclared](#)

- 7.6: QA Checklist Before Publishing - *Undeclared*
- 8: Using AI Agents to Write WeBWorK - *Undeclared*
 - 8.1: What are AI Agents? - *CC BY 4.0*
 - 8.2: Knowledge Documents for AI Agents - *Undeclared*
 - 8.3: Connecting AI Agents to the PG Renderer - *Undeclared*
 - 8.4: Advance AI Agents Advice - *Undeclared*
- 9: Appendices - *Undeclared*
 - 9.1: Minimal Templates - *Undeclared*
 - 9.2: Troubleshooting Checklist - *Undeclared*
 - 9.3: Macro Demonstrations - *Undeclared*
- Back Matter - *CC BY 4.0*
 - Index - *CC BY 4.0*
 - Glossary - *CC BY 4.0*
 - Detailed Licensing - *CC BY 4.0*
 - Detailed Licensing - *Undeclared*