The Programming Exercise Markup Language: Towards Reducing the Effort Needed to Use Automated Grading Tools

Divyansh S. Mishra
divyanshmishra@vt.edu
Virginia Tech
Blacksburg, VA, USA

Stephen H. Edwards
edwards@cs.vt.edu
Virginia Tech
Blacksburg, VA, USA

ABSTRACT

Automated programming assignment grading tools have become integral to CS courses at introductory as well as advanced levels. However such tools have their own custom approaches to setting up assignments and describing how solutions should be tested, requiring instructors to make a significant learning investment to begin using a new tool. In addition, differences between tools mean that initial investment must be repeated when switching tools or adding a new one. Worse still, tool-specific strategies further reduce the ability of educators to share and reuse their assignments. This paper describes an early experiences with PEML, the Programming Exercise Markup Language, which provides an easy to use, instructor friendly approach for writing programming assignments. Unlike tool-oriented data interchange formats, PEML is designed to provide a human friendly authoring format that has been developed to be intuitive, expressive and not be a technological or notational barrier to instructors. We describe the design and implementation of PEML, both as a programming library and also a public-access web microservice that provides full parsing and rendering capabilities for easy integration into any tools or scripting libraries. We also describe experiences using PEML to describe a full range of programming assignments, laboratory exercises, and small coding questions of varying complexity in demonstrating the practicality of the notation. The aim is to develop PEML as a community resource to reduce the barriers to entry for automated assignment tools while widening the scope of programming assignment sharing and reuse across courses and institutions.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**; • **Social and professional topics** → **Computing education**; • **Software and its engineering** → **Markup languages**.

KEYWORDS

programming assignment, automated grading, web service, notation, markup language, interchange format

ACM Reference Format:
Divyansh S. Mishra and Stephen H. Edwards. 2023. The Programming Exercise Markup Language: Towards Reducing the Effort Needed to Use Automated Grading Tools. In *Proceedings of the 54th ACM Technical Symposium on Computing Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569734

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569734

1 INTRODUCTION

Automated grading tools are becoming an increasingly common resource for educators, both as a way to cope with increasing student enrollments and as a mechanism to provide immediate and repeatable feedback on the syntax and run-time behavior of solutions. However, as new adopters explore possible tools, they find that each tool uses separate strategies for how assignments are specified, how behavioral tests are described, and in many cases even what styles of assignments are supported. Further, once one sets up an assignment for one system, the configuration and description is not very portable if one changes platforms or tools as needs evolve.

Further, many tools require direct interaction through a user interface to set up assignments, without any facilities for uploading or downloading assignment descriptions. Other tools use docker images, requiring educators to know (or learn) how to use docker to produce assignment configurations. These choices often make writing and maintaining assignment descriptions more time-consuming and involved than necessary, while also restricting the ability of instructors to share their assignments with colleagues, edit or modify assignments across semesters, or maintain their own internal collections of assignments they can adapt and reuse.

To address these issues, we describe PEML, the Program Exercise Markup Language. PEML provides a human-friendly authoring format for describing programming assignments, while also providing an easy path for integration into automated tools. It is an easy-to-learn, expressive, and robust way to write and maintain assignment descriptions and accompanying test suites. We also describe PEML's implementation through:

(1) A **PEML parser** library implemented in Ruby is publicly available as an open-source resource [14].
(2) A **web service** uses a REST API to provide programming-language-independent access to PEML parsing services [13]. The API provides all the features of the parsing library, including client access to JSON, YAML, or XML representations of parsed contents to promote easier integration with existing tools or scripting languages. The web service supports HTML rendering of assignment instructions and executable software tests that are "ready to use" for grading tools.
(3) A **user-facing website** that serves as a "PEML playground" for experimenting with PEML provides direct user-level access to JSON, YAML, or XML results of parsed descriptions just like the tool-oriented REST API [12].

This paper describes the design of PEML, explaining how it achieves its goal of being author-friendly and easy to learn. We also describe our early experiences validating PEML's utility. We explored practical usage of PEML by randomly selecting 60 exercises from the CS1 course at Virginia Tech . The exercises spanned a range of sizes, complexities, and purposes. We included short programming practice exercises used in homework, medium-sized laboratory assignments that involved writing a small number of classes in a few hours, and out-of-class programming assignments that were intended to take days or a week of individual work writing complete programs. All assignments were then represented in PEML to demonstrate it could express this range of activities, passed through the parser to ensure its functioning, and checked the resulting rendered executable tests produced from the descriptions. Finally, we describe the implementation of PEML and the web service that provides easy access for tools in any programming language to integrate PEML descriptions into their processes.

2 LITERATURE REVIEW

A central focus of our work is to allow assignment authors to be able to use PEML directly with automated grading tools by making it streamlined for tool developers to incorporate PEML support. This section provides background on automated grading tools, their design specifications, design specifications for testing, and data interchange formats that could be employed.

2.1 Automated Grading Tools

Automated grading tools [10, 20] have allowed instructors of CS courses to focus more on the instructional aspects and on helping students by automating a large stretch of the assignment feedback pipeline. Ihantola et al. [9] provide a survey of many well-known tools where they talk about the development context for some of them as well as common features provided including testing, grading, manual assessment, and submission policies. They conclude with describing a rise in number of new tools being developed, as well as providing recommendations on areas newer tools should focus on. Nayak et al. [15] provide a more recent survey conducted on well-known automated grading tools where they discuss about features such as type of analysis, feedback generation, grading methodologies, and degree of manual involvement. They also introduce an ideal automatic assessment model that would remedy certain issues current automated grading tools suffer from.

With the help of automated grading tools, instructors are able to provide instant feedback [4, 6], the nature of which, depends on the choice of tool, its grading approaches, assessment types, and any customization conducted. Feedback from such tools benefits students in many ways [6, 7]. While many such tools are in current use, the focus of this paper is supporting a human-friendly and tool-friendly assignment description format that is tool-neutral, so we do not focus on the use of a specific auto-grader.

2.2 Design Specification for Automated Grading

Automated grading tools tend to be based around certain design specifications which govern the formats of programming assignments, test-suites, and graders [5]. The most commonly used formats are JSON and YAML with a wide variety of further design constraints in each depending upon the nature of the tool [8]. These can range from support for in-lined external files and support for markdown/HTML all the way to tool specific key-value pair representation of entire assignment descriptions.

Agrawal and Reed [1] provide a survey of grading formats used by well-known automated grading tools. They discuss the prevalence of both tool-specific and JSON-like notations. While it may require additional development effort for a tool to parse PEML descriptions, using formats like JSON and YAML requires less work.

2.3 Design Specification for Test Generation

Automated grading tools often use test cases to grade submitted assignments [17]. Test suites can be part of the tool-specific formal representation of the assignment or be provided as separate input. Some tools use x-unit style unit tests expressed in programming code [3]. Other tools define tests using pairs of given and expected values. PEML supports both of these approaches. Test cases can be embedded directly into an assignment description or provided as separate files, using general-purpose or tool-specific formats.

2.4 Data Interchange Formats

When thinking of possible input formats for automated tools, it is natural to think of existing data interchange formats. There are several commonly used data interchange formats that are well-specified and widely supported. However, while nicely supported by tools, they often are less friendly for human authoring.

JSON [16] is perhaps the most commonly used interchange format at present. It is language-independent and uses JavaScript syntax. It is based on two types of data structures, a collection of key/value pairs known as a hash, map, dictionary, or struct, and an ordered sequence of values known as an array, vector, or list.

YAML [19] is another well-known data interchange format that is commonly used for writing configuration files. It uses python style indentation to represent nesting. More recently, it has incorporated JSON as a subset.

XML [18] is an older interchange format that evolved from SGML. It has similarities with HTML, albeit with stricter rules and higher verbosity. However, XML is less popular in modern applications because of its verbosity and the more cumbersome nature of its parsing and validation libraries, compared to newer alternatives.

Unfortunately, these existing formats are not writer-friendly enough for descriptions that contain large amounts of free-form text. Programming activities usually require sizeable chunks of multi-line text to describe their properties, whether it is the specification for a program, or starter code to provide to a student, or reference tests to check a solution, or a sample/reference solution to provide for other instructors to look at. In most cases, describing assignments is not focused on simple key/value pairs where values are small pieces of data, or about deeply structured nested object descriptions. It is about writer-friendly input of structured text where most of the values are multi-line text written by humans.

As a result, the most common data interchange formats in use today are not syntax-friendly enough to present a minimal-effort entry path for new authors. Of course, this is not an obstacle for programming-oriented instructors who are already familiar with YAML or JSON. However, those formats do require a learning curve

that we hope to minimize further, while also leading to syntax errors in markup. In particular, YAML's reliance on whitespace and indentation to indicate nested structure can be an obstacle for free-form text input. JSON's use of JavaScript quoting and lack of true multi-line values makes it challenging in similar ways.

Instead, PEML takes significant inspiration from ArchieML [2], an interchange format that originated at the New York Times for non-programmers to write and edit structured text. One of their goals was to make it easy to write structured text "without having [to] type a lot of special characters" in a form that "makes sense to non-programmers". At the same time, however, ArchieML can express exactly the same nested, structured data built from dictionaries and arrays as other notations.

3 THE DESIGN OF PEML

The Programming Exercise Markup Language (PEML) has been designed to support assignment authors by streamlining common use cases to provide a low-energy onboarding path for authors.

PEML uses a plain-text representation for describing exercises inspired by ArchieML. PEML, like YAML and JSON, uses a key-value structure to describe programming assignments. The keys used are alphanumeric identifiers, to simplify parsing and increase writeability. PEML also supports the use of the period character to represent nested keys, as in ArchieML, which allows for easier nesting compared to indentation-based nesting in YAML and JavaScript object-based nesting in JSON. Keys start at the beginning of the line, and, for keys with single values, end with a colon. Values follow after the colon and can extend to the beginning of the next property. All leading/trailing whitespaces are trimmed.

Like YAML, PEML uses the '#' character for comments, which must appear on lines by themselves. PEML also uses a specific comment line, using '#—' to signal the start of a PEML description. While this is optional for the first PEML description, it serves as a delimiter for subsequent PEML descriptions when multiple exercises are included in the same file or stream.

```
exercise_id: https://cssplice.github.io/peml/examples/01-minimal.peml

# This is a minimalist example of the barest properties needed for
# an exercise description. It includes a license instead of author
# (you can provide both, if desired)

# title is required
title: A Minimal PEML Description

# Listing the license is very strongly encouraged, but not mandatory
# See: https://cssplice.github.io/peml/schemas/exercise.html#license
# For id's, see keywords at:
# https://help.github.com/en/articles/licensing-a-repository
license.id: cc-sa-4.0
license.owner.email: edwards@cs.vt.edu
license.owner.name: Stephen Edwards

# Must include at least one of: instructions or systems or suites
instructions:----------
Write instructions for your exercise here.
----------
```

Figure 1: A basic PEML description.

PEML also supports quoting using a variant of the HereDoc style, similar to triple quoting in Python and Scala. Any key that is followed by three or more repetitions of the same printing character (forming a custom delimiter) is treated as a HereDoc style quoted value. This is more flexible than triple quoting which itself might appear in assignment descriptions with code fragments. This also improves upon how JSON and YAML handle multi-line free-form text. The quoted value is terminated by the first subsequent occurrence of the custom delimiter pattern.

PEML also supports textual descriptions written in markdown as well as embedding HTML directly into exercise descriptions. Optional use of other text-based markup formats is also possible, although library users providing PEML implementations may not support all possible formats. The PEML reference implementation and web service API also support automatic conversion (rendering) of markdown text embedded in exercise descriptions into HTML for client-level use.

PEML allows referencing of external resources in two ways: first, any value can be provided using an external reference rather than inlining it by using the **url(...)** construct (similar to its use in CSS). The URL can be absolute, referring to a resource on the web, or relative, referring to local files that are bundled with the PEML description, such as, in a single zip file. The second way is based on the idea of convention-over-configuration, where external resources like images and libraries are put under the "public_html/" folder that is located alongside the PEML description in the same folder, zip file, or repository. Within Markdown or HTML keys, relative URLs that start with "public_html/..." will then be correctly resolved to these resources.

PEML also has support for authors to provide parts of PEML description itself from external files using the ':include' directive, which fetches PEML descriptions from external locations. While not required, it can be used to factor out repeated key/value pairs to aid reusability. This feature also allows authors to separate out test cases and test environments from the main exercise description, placing them in separate files as desired, in order to control access or visibility while allowing the main PEML description to be publicly available. This feature improves over YAML and JSON which both lack constructs like 'import' and 'include' and depend on the tool/reader to fetch information from other sources.

PEML also allows authors to write parameterized exercises where many instances can be produced using different parameter values for different students. This type of string interpolation is provided using "mustache notation" ('{{}}'). Authors can use any number of user-defined variables, where occurrences of variable_name are replaced by provided values. While PEML does not support escaping literals '{{' and '}}', authors can specify a different set of delimiters using the "options.interpolation. delimiter" key or specify variable names that should not be interpolated in the "options.interpolation.exclude" field. Again, this feature improves upon the lack of string interpolation using an alias in YAML and JSON.

Beyond single-value keys, PEML supports nested structures based on ArchieML's convention for dotted keys, object blocks, and arrays. As with ArchieML, arrays are defined with keys enclosed in '[...]' brackets and terminated with an empty pair of brackets '[]'.

```
exercise_id: edu.vt.cs.cs1114.palindromes

# Single-line comments start with #
# Comments must be on lines by themselves

title: Palindromes (A Simple PEML Example)

license.id: cc-sa-4.0
license.owner.email: edwards@cs.vt.edu
license.owner.name: Stephen Edwards

topics: Strings, loops, conditions
prerequisites: variables, assignment, boolean operators

instructions:----------
Write a program that reads a single string (in the form of one line
of text) from its standard input, and determines whether the string is
a _palindrome_. A palindrome is a string that reads the same way
backward as it does forward, such as "racecar" or "madam". Your
program does not need to prompt for its input, and should only generate
one line of output, in the following
format:

```
"racecar" is a palindrome.
```

Or:

```
"Flintstone" is not a palindrome.
```
----

assets.test_format: stdin-stdout

[systems]
language: java
version: >= 1.5
[]

Test data/files/classes are probably located in separate files, but this
is a simple example of how to do something directly inline
[suites]
[.cases]

stdin: racecar
stdout: "racecar" is a palindrome.

stdin: Flintstone
stdout: "Flintstone" is not a palindrome.

stdin: url(some/local/input.txt)
stdout: url(some/local/output.txt)

stdin: url(http://my.school.edu/some/local/generator/input)
stdout: url(http://my.school.edu/some/local/generator/output)

[]
[]
The [] ends/closes a list of items, but they can be omitted at the
end of the file, since EOF auto-closes any open lists. The first []
closes the list of cases in one suite, while the second [] closes
the list of suites, which here includes only one suite.
```

Figure 2: PEML description of a short palindrome exercise.

Any trailing empty brackets or braces at the end of the file can be omitted. PEML uses repeated occurrences of the first key in any array item to mark the start of a new item in that array, so the first key should be used consistently to mark the start of a new item. Data structured in PEML can be arbitrarily nested with the use of dotted keys as discussed above.

4 PEML'S IMPLEMENTATION

The following subsections correspond to the implementation approaches automated grading tools can take to integrate PEML descriptions in their processes. These implementation approaches cover the PEML parser, the PEML REST service, and the PEML Rails application respectively.

4.1 PEML Parser

PEML has been designed with the goal of providing authors of programming assignments with a human-friendly way to describe assignments while also supporting automatic grading based on PEML descriptions. We built a parser for PEML which allows PEML descriptions to be converted into JSON, YAML, or XML descriptions for easy consumption by educational tools.

The PEML parser is written in Ruby and is publicly available as a gem [14]. The main entry point for the gem is the 'Peml.parse()' function, which takes a PEML representation in string form or as a file. The function returns a hash of two key/value pairs: a nested hash representation of the parsed PEML contents, together with an array of any generated diagnostics.

The parser supports optional arguments to control additional processing of the parsed PEML content. The 'interpolate' flag can be set to signal the parser to substitute variables embedded in PEML field values following the mustache-style notation discussed earlier. The 'render_to_html' flag can be set to convert descriptions written in markdown into HTML in the returned data representation.

4.2 A PEML Web Service

Of course, not all automatic grading tools will be written in Ruby and be able to utilize the parser as a gem. Developing multiple libraries to support an array of programming languages requires substantial effort. Instead, we also provide the parser's features through a web service using a REST API [13] implemented using a Ruby on Rails application. Clients can use HTTP POST (or GET) requests to the parsing service, providing a PEML string or file as a parameter, as well as specifying any optional parameter values. The client can request to receive the parsed results in JSON, YAML, or XML format, to best suit their needs. Since the web service is built around the parser, it offers all features provided by the gem.

4.3 PEML Live!

The web application providing the parsing API also provides a human-oriented web user interface that allows users to write PEML

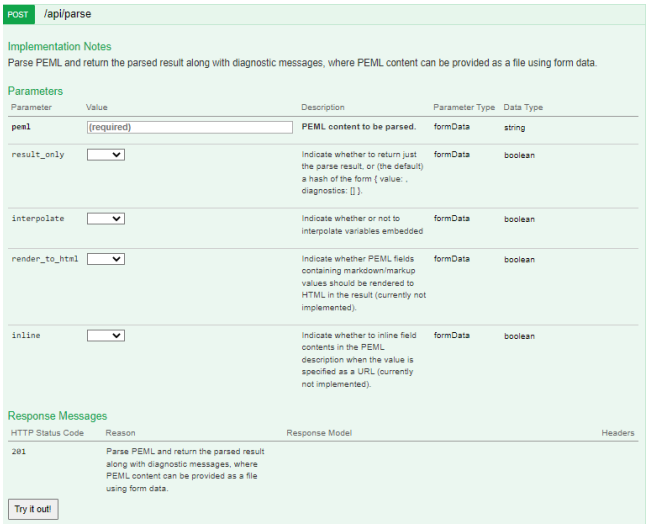

Figure 3: REST API documentation on the PEML website.

descriptions and parse them directly in their web browser [12]. It also provides a set of pre-existing PEML descriptions for users to start with. Users can select from a set of additional features as well as a preferred response format when parsing their PEML descriptions. Users can see the results of the parser live, right in the browser.

The application also provides additional details about PEML such as its goals and design decisions, its specifications, documentation on the REST API, and a diverse set of PEML examples for getting started with writing PEML descriptions.

5 EVALUATION AND EXPERIENCE

In this section, we discuss our experience working with PEML, the PEML parser, as well as the PEML web Service. In the following subsections, we touch upon our method for this part of the paper, our experience converting existing assignment descriptions to PEML, our experiences working with the parser, and finally, the web service, particularly its ability to generate executable test cases for use in automated grading tools.

5.1 Method

To evaluate the feasibility of using PEML, we generated PEML descriptions for a collection of existing CS1 assignments. This involved choosing which assignments to work with and then writing and reviewing PEML descriptions for each one. Our goal was to cover a broad range of programming topics while also representing a variety of classroom programming activities. We also wanted a range of difficulty levels as well as styles within each of the topics selected. Lastly, we wanted a good balance between assignments that used pre-written unit tests and those using data-driven variable-substitution based techniques for describing test cases.

We selected 60 problems from three kinds of activities. We chose 50 short-answer practice exercises, which most students answer in a few minutes each as part of homework. We also selected 5 lab assignments, which take a few hours each. Then we added 5 larger out-of-class programming assignments, which students work on for several days or a week. All were taken from the CS1 course at our university. All PEML descriptions from this study are publicly available [11]. Table 1 shows examples from the problems selected.

5.2 Experience Writing PEML Descriptions

When encoding existing assignment descriptions into PEML using text editing tools, the first few were created without any syntax highlighting in a plain text editor. This can feel visually jarring coming from using IDEs or other programming tools that provide color highlighting and syntax suggestions, to which many of us have become accustomed. This kind of support is readily available for other formats like JSON and YAML. Fortunately, it was straightforward to create a custom editing mode definition for tools like BBEdit and SublimeText that supports PEML.

Another aspect of PEML that stood out during the encoding phase was how reusable the templates were. Most assignments at the CS1 level follow a structure having assignment instructions, some starter or wrapper code, test cases (at least, for assignments where auto-grading is used), and finally, course specific identifiers. These aspects of the assignments can be provided inline, as references to external file locations, or using URLs. PEML's use of "convention over configuration" made it easy to have a generic PEML template where new assignments can be generated just by plugging in new instructions, test cases, and starter and wrapper code inline, or by using a predefined directory structure. This streamlined the generation of PEML descriptions so that, after completing a few, it became quick and easy to do many more.

It was also useful that the PEML parser renders markdown to HTML. This really comes in handy when using PEML descriptions for online tutoring and grading tools that provide students with an IDE-like environment to solve questions and receive feedback. Instructions can be provided using git-flavor markdown and the PEML parser will automatically convert it to HTML for use in grading tools. Another helpful feature with PEML is on-the-fly variable interpolation that allows authors to generate multiple versions of the same assignment with key values changed.

5.3 PEML Errors and Diagnostics

As with any notation, writing new documents produces occasional syntax errors. The PEML parser returns the parsed description along with a list of any diagnostic messages generated. Unlike typical programming languages, true syntax errors are nearly absent, as in ArchieML. Instead, diagnostics result from validating the data representation, identifying required keys that are absent or improperly structured nested fields. While performing our feasibility study, we recorded and evaluated all diagnostic messages received.

The most common diagnostic message pointed to a missing author key. This was due to the fact that a group of the short programming exercises we used had their author names omitted. Another common diagnostic message identified externally referenced files that were missing, which was due to them being out of place. Finally, a less frequently occurring message was one about strings in test cases being incorrectly formatted when test input/output

Name	Description
Hello Cat	A “hello world” style main program that produces fixed text output
Modeling a Microwave Oven	Writing multiple classes, using inheritance to create subclasses
The Pig Game	Implementing a multi-round dice game with a text user interface and computerized opponent
Talk Like a Pirate	A text I/O program that uses string substitutions to translate input strings into pirate talk
Shape Maker	A graphical drawing editor with a Java Swing user interface
Mine Sweeper	Using a two-dimensional array to represent a game board for a provided graphical user interface
Time Table	A lab assignment practicing with two-dimensional arrays
Counting Lines	A lab assignment practicing text input using scanners
Building a Gradebook	Practice with maps and lists
Spelling Checker	A lab assignment practicing using maps and text input
Rot-13 Decryption	A lab assignment practicing string manipulation
arrayListCodingPractice	small exercise practicing with lists
arraysFindOnes	small exercise practicing with one-dimensional arrays
binarySearchNonRecursive	small exercise practicing binary search
callingSuperPractice2	small exercise practicing using ’super’
genericsComparableCage.peml	small exercise practicing using generics
implementAConstructor	small exercise practicing writing constructors
	[44 other small exercises not shown]

Table 1: A subset of the 60 assignments used in the feasibility study.

values were provided as traditional CSV files, where quoting values by hand can be error-prone. PEML provides an alternate tabular format for data-driven descriptions of test cases that is more human-friendly than CSV—it allows full programming language expressions and quoting styles. By receiving these diagnostics messages immediately upon parsing, we were able to rectify any errors pointed out and fix the descriptions before proceeding.

5.4 Unit Testing Assignments

We evaluated the ability of the parser to generate unit test cases that can be used to grade student submissions. Since the assignments used in our feasibility study were already part of a pre-existing CS1 course, they came with embedded unit tests. Some exercises came with software tests written in JUnit, while many small programming exercises instead had test cases specified using input and expected output values in columns of CSV data. Many automated grading tools expect tests to be specified using input/output pairs in some form—even if these are strings provided as standard input to a main program or collected from the program’s standard output stream. PEML allows descriptions of unit tests in tabular form with a flexible and extensible column format, where that is appropriate. PEML uses a simple templating scheme to automatically generate executable software tests from tabular descriptions in a way that is programming-language-agnostic. We tested this approach with Java, Python, and C++ to evaluate its feasibility. By providing customized templates to the parser as parameters, autograding tools can render data-driven tests in tool-specific ways if desired.

We setup a script to parse the PEML descriptions using the REST service and then extracted the generated executable software tests from the result. These test cases are generated through variable interpolation of values from the inlined data table into language-specific Liquid templates. We then compiled the tests to ensure they were rendered successfully. Our aim here is to measure if all the test cases compile and run and if we are able to validate solutions in a way that a manual grader would.

6 CONCLUSION AND FUTURE WORK

PEML has been specifically designed to provide instructors with a human-friendly way to write, maintain, and reuse assignment descriptions that can be directly used by auto-grading tools. The parser library [14] and the REST API [13] have been designed to aid the integration of PEML into existing automated grading pipelines with minimal effort, and the PEML website provides instructors a sandbox environment to learn and experiment with PEML [12].

Our results show that writing PEML descriptions is easier than using other data-interchange formats. We also show how the parser can provide feedback on description quality and how it can generate test cases, which we checked for compilability and validity.

In the future, we plan on expanding our feature-set for the REST API as well as providing a DSL for writing human-friendly test cases natively inside PEML descriptions. In addition, we are now hard at work with two automated grading tools to integrate PEML support so that instructors can begin using PEML as a way to enter or upload their assignment descriptions. As tools begin to support PEML, the goal of easing the required effort needed to adopt automated grading tools will become more achievable.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DRL-1740765. 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.

REFERENCES

[1] A. Agrawal and B. Reed. 2022. A SURVEY ON GRADING FORMAT OF AUTOMATED GRADING TOOLS FOR PROGRAMMING ASSIGNMENTS. In *ICERI2022 Proceedings* (Seville, Spain) *(15th annual International Conference of Education, Research and Innovation)*. IATED, 7506–7514. https://doi.org/10.21125/iceri.2022.1912

[2] ArchieML. 2022. *Archie Markup Language.* Retrieved August 15, 2022 from http://archieml.org/

[3] Christopher Brown, Robert Pastel, Bill Siever, and John Earnest. 2012. JUG: A JUnit Generation, Time Complexity Analysis and Reporting Tool to Streamline Grading. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 99–104. https://doi.org/10.1145/2325296.2325323

[4] Stephen H. Edwards. 2021. Automated Feedback, the Next Generation: Designing Learning Experiences. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 610–611. https://doi.org/10.1145/3408877.3437225

[5] Stephen H. Edwards, Jürgen Börstler, Lillian N. Cassel, Mark S. Hall, and Joseph Hollingsworth. 2008. Developing a Common Format for Sharing Programming Assignments. *SIGCSE Bull.* 40, 4 (nov 2008), 167–182. https://doi.org/10.1145/1473195.1473240

[6] Xiang Fu, Boris Peltsverger, Kai Qian, Lixin Tao, and Jigang Liu. 2008. APOGEE: Automated Project Grading and Instant Feedback System for Web Based Computing. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 77–81. https://doi.org/10.1145/1352135.1352163

[7] Marcelo Guerra Hahn, Silvia Margarita Baldiris Navarro, Luis De La Fuente Valentín, and Daniel Burgos. 2021. A Systematic Review of the Effects of Automatic Scoring and Automatic Feedback in Educational Settings. *IEEE Access* 9 (2021), 108190–108198. https://doi.org/10.1109/ACCESS.2021.3100890

[8] Aliya Hameer and Brigitte Pientka. 2019. Teaching the Art of Functional Programming Using Automated Grading (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 115 (jul 2019), 15 pages. https://doi.org/10.1145/3341719

[9] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[10] Alan Marchiori. 2022. Labtool: A Command-Line Interface Lab Assistant and Assessment Tool. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3478431.3499285

[11] Divyansh S. Mishra and Stephen H. Edwards. 2022. *PEML Examples.* Retrieved December 14, 2022 from https://github.com/CSSPLICE/peml-feasibility-examples

[12] Divyansh S. Mishra and Stephen H. Edwards. 2022. *PEML Live!* Retrieved December 14, 2022 from https://cssplice.github.io/peml/peml-live.html

[13] Divyansh S. Mishra and Stephen H. Edwards. 2022. *PEML REST API.* Retrieved December 14, 2022 from https://cssplice.github.io/peml/peml-api.html

[14] Divyansh S. Mishra and Stephen H. Edwards. 2022. *PEML: The Programming Exercise Markup Language.* Retrieved December 14, 2022 from https://github.com/CSSPLICE/peml

[15] Sidhidatri Nayak, Reshu Agarwal, and Sunil Kumar Khatri. 2022. Automated Assessment Tools for grading of programming Assignments: A review. In *2022 International Conference on Computer Communication and Informatics (ICCCI)*. 1–4. https://doi.org/10.1109/ICCCI54379.2022.9740769

[16] JavaScript Object Notation. 2022. *Introducing JSON.* Retrieved August 15, 2022 from https://www.json.org/

[17] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616

[18] XML. 2022. *Extensible Markup Language.* Retrieved August 15, 2022 from https://www.w3.org/XML/

[19] YAML 1.2. 2022. *YAML Ain't Markup Language.* Retrieved August 15, 2022 from https://yaml.org/

[20] Jeremy K. Zhang, Chao Hsu Lin, Melissa Hovik, and Lauren J. Bricker. 2020. GitGrade: A Scalable Platform Improving Grading Experiences. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1284. https://doi.org/10.1145/3328778.3372634